

ENCYCLOPEDIA OF COMPUTER SCIENCE

ANTHONY RALSTON, Editor
CHESTER L. MEEK, Assistant Editor

FIRST EDITION



PETROCELLI / CHARTER

NEW YORK 1976

Copyright Mason/Charter Publishers, Inc. 1976

All rights reserved. No part of this work covered by the copyrights hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, or taping, or information storage and retrieval systems—without written permission of the publisher.

First Printing

Printed in the United States of America

Library of Congress Cataloging in Publication Data

Main entry under title:

Encyclopedia of computer science,

Includes index.

1. Computers --Dictionaries .2. Electronic data processing--Dictionaries. 3. Information science--Dictionaries. I, Ralston, Anthony, II, Meek, Chester L., 1941-

QA76.15.E48

001.6'4'03

76-15032

ISBN 0-88405-321-0

CONTENTS

Editorial Board	vii
Contributors	vii
Preface	xi
Editor's Foreword	
Organization	xiii
Using the Encyclopedia	xiv
Classification of Articles	xvii
Encyclopedia	1-1464
Appendixes	
I. Abbreviations and Acronyms, Mathematical Notation, Units of Measure	1465
II. Useful Numerical Tables	1475
Index	1479

EDITORIAL BOARD

John M. Bennett
University of Sydney
A. S. Douglas
*The London School of Economics and
Political Science*
Nicholas V. Findler
State University of New York at Buffalo
Aaron Finerman
State University of New York at Stony Brook
Patrick C. Fischer
Pennsylvania State University
C. C. Gotlieb
University of Toronto

John A. N. Lee
Virginia Polytechnic Institute
Saul Rosen
Purdue University
Gerard Salton
Cornell University
Andries van Dam
Brown University
Eric A. Weiss
Sun Oil Company

CONTRIBUTORS

John N. Ackley, *Digital Computer Controls, Inc.*
A. K. Agrawala, *University of Maryland*
Suad Alagić, *Univerzitet U. Sarajevu*
Saul Amarel, *Rutgers University*
J. K. Amsbaugh, *Sun Oil Co.*
Isaac L. Auerbach, *Auerbach Corporation*
Algirdas Avitienis, *University of California at
Los Angeles*
Charles W. Bachman, *Honeywell Corp.*
G. David Baer, *Computer Task Group, Inc.*
Jean-Loup Baer, *University of Washington*
D. W. Barron, *University of Southampton, U.K.*
F. L. Bauer, *Technischen Universität, Munich*
John M. Bennett, *University of Sydney*
Gilbert R. Berglass, *State University of New York at
Buffalo*
John C. G. Boot, *State University of New York at
Buffalo*
A. D. Booth, *Lakehead University*

K. H. V. Booth, *Lakehead University*
Susan Brewer, *Honeywell Corp.*
Peter J. Brown, *University of Kent, U.K.*
William F. Brown, *Ball State University*
G. Edward Bryan, *Xerox Corp.*
Janusz A. Brzozowski, *University of Waterloo*
W. Buchholz, *IBM Corp.*
Sander Buchman, *IBM Corp.*
R. A. Buckingham, *University of London*
John Case, *State University of New York at Buffalo*
Robert P. Cervený, *State University of New York at
Buffalo*
Ned Chapin, *InfoSci, Inc.*
W. J. Cody, *Argonne National Laboratory*
Samuel D. Conte, *Purdue University*
James W. Cooley, *IBM Corp.*
David C. Cooper, *University of London*
Joseph F. Cunningham, *Boca Raton, Fla.*
B. Garland Cupp, *McDonnell-Dough Automation Co.*

CONTRIBUTORS

- Charles H. Davidson, *University of Wisconsin*
 Dorothy E. Denning, *Purdue University*
 Peter J. Denning, *Purdue University*
 George D. Detlefsen, *General Electric Co.*
 J. K. Dixon, *Naval Research Laboratory*
 T. A. Dolotta, *Bell Laboratories*
 Philip H. Dorn, *Dorn Computer Consultants*
 A. S. Douglas, *London Sch. of Economics & Polit. Science*
 Michael A. Duggan, *University of Texas*
- Patricia James Eberlein, *State University of New York at Buffalo*
 Richard H. Eckhouse, Jr., *Digital Equipment Corp.*
 Carl Engelman, *The MITRE Corp.*
 Kurt Enslein, *Genesee Computer Center, Inc.*
- B. R. Faden, *North American Rockwell*
 Jerome A. Feldman, *University of Rochester*
 Nicholas V. Findler, *State University of New York at Buffalo*
 Aaron Finerman, *State University of New York at Stony Brook*
 Clive B. Finkelstein, *IBM Australia*
 Patrick C. Fischer, *Pennsylvania State University*
 Dennis A. Fletcher, *D. A. Fletcher & Associates*
 Ivan Flores, *Baruch College, CUNY*
 Michael J. Flynn, *Stanford University*
 Caxton C. Foster, *University of Massachusetts*
 Mark A. Franklin, *Washington University*
 David N. Freeman, *Ketron, Inc.*
 Gideon Frieder, *State University of New York at Buffalo*
 Samuel H. Fuller, *Carnegie Mellon University*
- Bernard A. Galler, *University of Michigan*
 W. Morven Gentleman, *University of Waterloo*
 Bruce Gilchrist, *Columbia University*
 Amos N. Gileadi, *University of Massachusetts*
 Stanley Gill (deceased)
 George Glaser, *Consultant*
 Jonathan Goldstine, *Pennsylvania State University*
 Robert H. Gonter, *University of Massachusetts*
 C. C. Gotlieb, *University of Toronto*
 Thomas S. Grier, *Burroughs Corp.*
 Michael D. Grigoriadis, *IBM Corp.*
 Ralph E. Griswold, *University of Arizona*
 Fred Gruenberger, *California State University at Northridge*
 Mark Halpern, *Tymshare, Inc.*
 John W. Hamblen, *University of Missouri at Rolla*
 R. W. Hamming, *Bell Laboratories*
 Fred H. Harris, *University of Chicago*
 Robert V. Head, *Consultant*
 L. B. Heilprin, *University of Maryland*
- Herbert Hellerman, *State University of New York at Binghamton*
 Harry A. Helm, *Defense Communications Agency*
 Gabor T. Herman, *State University of New York at Buffalo*
 I. T. Ho, *IBM Corp.*
 Richard C. Holt, *University of Toronto*
 John E. Hopcroft, *Cornell University*
 J. N. P. Hume, *University of Toronto*
 E. Gerald Hurst, Jr., *University of Pennsylvania*
 Harry D. Huskey, *University of California at Santa Cruz*
 S. R. Hyde, *Joint Speech Research Unit, U.K.*
- R. V. Jacobson, *Chemical Bank, New York*
 Charles V. Jones, *York University*
 T. L. Jones, *Howard University*
- Laveen N. Kanal, *University of Maryland*
 Arthur I. Karshmer, *University of Massachusetts*
 Ronald H. Kay, *IBM Corp.*
 Kenneth M. Kempner, *National Institutes of Health*
 Robin H. Kerr, *General Electric Co.*
 Michael J. Kessler, *Control Data Corp.*
 Peter T. Kirstein, *University of London*
 Kenneth E. Knight, *University of Texas*
 Robert R. Korfhage, *Southern Methodist University*
 David J. Kuck, *University of Illinois*
 Shan S. Kuo, *University of New Hampshire*
- Börje Langefors, *University of Stockholm*
 Duncan H. Lawrie, *University of Illinois*
 Charles L. Lawson, *California Institute of Technology*
 John A. N. Lee, *Virginia Polytechnic Institute*
 R. P. Leinius, *Union Carbide Corp.*
 Arthur Llewelyn, *Computer-Aided Design Centre, Cambridge, U.K.*
 Keith R. London, *Keith London Ltd., U.K.*
 Harold Lorin, *IBM Corp.*
 Col. William F. Luebbert, *U.S. Military Academy*
 Daniel H. Lufkin, *Consultant*
- George Marsaglia, *McGill University*
 Francis F. Martin, *Consultant*
 Johannes J. Martin, *Virginia Polytechnic Institute*
 Francis Parkash Mathur, *University of Missouri at Columbia*
 David W. Matula, *Southern Methodist University*
 Michael M. Maynard, *Sperry Rand Corp.*
 Davis B. McCarn, *National Library of Medicine*
 John McCarthy, *Stanford University*
 Edward J. McCluskey, *Stanford University*
 Daniel D. McCracken, *Consultant*
 John M. McKinney, *University of Cincinnati*

CONTRIBUTORS

- John C. McPherson, *Amagansett, New York*
 C. L. Meek, **Andco, Inc.**
 Albert R. Meyer, *Massachusetts Institute of Technology*
 Benjamin Mittman, *Northwestern University*
 Georgia Mollenhoff, *Journalist*
 Calvin N. Mooers, *Rockford Research, Inc.*
 Bruce C. Moore, *Louisiana State University*
 G. J. Morris, *International Computers, Ltd.*
 Philip M. Morse, *Massachusetts Institute of Technology*
 François Muller, *United Nations*
 Jean E. Musinski, *Cornell University*
 J. Necas, *Prague, Czechoslovakia*
 Roger M. Needham, *University of Cambridge, U. K.*
 Allen Newell, *Carnegie-Mellon University*
 Carol M. Newton, *University of California at Los Angeles*
 Jerre D. Noe, *University of Washington*
 T. William Olle, *Consultant*
 Leonard J. Palmer, *Palmer Data Corporation*
 Donn B. Parker, *Stanford Research Institute*
 Azaria Paz, *Technion-Israel Institute of Technology*
 Trevor Pearcey, *Caulfield Institute of Technology, Australia*
 C. R. Pearson, *Georgia Institute of Technology*
 Milton Pine, *Caulfield Institute of Technology, Australia*
 Seymour V. Pollack, *Washington University*
 Michael J. F. Poulsen, *IBM Australia*
 C. E. Price, *Union Carbide Corp.*
 Anthony Ralston, *State University of New York at Buffalo*
 Brian Randell, *University of Newcastle upon Tyne*
 Bertram Raphael, *Stanford Research Institute*
 Edwin D. Reilly, Jr., *State University of New York at Albany*
 Lee Revens, **ACM**
 John R. Rice, *Purdue University*
 Frederic N. Ris, **IBM Corp.**
 Saul Rosen, *Purdue University*
 Azriel Rosenfeld, *University of Maryland*
 Robert F. Rosin, *Iowa State University*
 Paul Roth, *National Bureau of Standards*
 B. C. Rowe, **Polytechnic of North London**
 Arthur I. Rubin, *Electronic Associates Inc.*
 Harry J. Saal, *IBM Israel*
 Arto Salomaa, *University of Aarhus, Denmark*
 David Salomon, *San Diego State University*
 Gerard Salton, **Cornell University**
 Jean E. Sammet, **IBM Corp.**
 Adel S. Sedra, *University of Toronto*
 Sally Yeates Sedelow, **University of Kansas**
 Eugene Shapiro, *IBM Corp.*
 Ben Shneiderman, *Indiana University*
 Herbert A. Simon, **Carnegie-Mellon University**
 James R. Slagle, *Naval Research Laboratory*
 Vladimir Slamecka, **Georgia Institute of Technology**
 Alvy Ray Smith, III, **Xerox Corp.**
 Cecil L. Smith, *Louisiana State University*
 I. A. Smith, *State University of New York at Binghamton*
 Kenneth C. Smith, *University of Toronto*
 John S. Sobolewski, *University of Washington*
 Fred A. Stahl, *Columbia University*
 Elizabeth Luebbert Stoll, *Santa Clara, Calif.*
 T. B. Steel, **Equitable Life Assurance Society**
 Theodor D. Sterling, *Simon Fraser University*
 Jon C. Strauss, *University of Pennsylvania*
 Robert W. Taylor, *University of Massachusetts*
 Daniel Teichroew, *University of Michigan*
 Thilo Tilemann, *University of Cologne*
 Anthony L. Torrance, **McKinsey & Company, Inc.**
 Henry S. Tropp, **Humboldt State University, California**
 D. C. Tschritzis, *University of Toronto*
 Andries van Dam, *Brown University*
 R. H. VanDenburg, Jr., *Southern Bank and Trust Co.*
 W. M. Waite, *University of Colorado*
 Charles H. Warlick, *University of Texas*
 Peter Wegner, *Brown University*
 Eric A. Weiss, **Sun Oil Company**
 Anne H. Werkheiser, *The MITRE Corp.*
 Milton R. Wessel, *Attorney*
 Gio Wiederhold, *University of California at San Francisco*
 Maurice V. Wilkes, *University of Cambridge, U.K.*
 James H. Wilkinson, *National Physical Laboratory, U.K.*
 D. Wotschke, *Pennsylvania State University*
 Stephen S. Yau, *Northwestern University*
 David M. Young, *University of Texas*
 H. Zemanek, *IBM Austria*
 Karl L. Zinn, *University of Michigan*
 Stanley Zions, *State University of New York at Buffalo*
 Albert L. Zobrist, *University of Arizona*

PREFACE

When the idea of an Encyclopedia of Computer Science was first proposed to me almost five years ago, I embraced it eagerly. I believed, then, and believe even more strongly now, that computer science has come sufficiently of age as a discipline that it is appropriate and necessary to produce-in breadth and in depth-a snapshot of it (for, after all, a snapshot is what an encyclopedia is). Equally important is a belief that such a snapshot will have value not just for the moment but for some considerable number of years. The discovery and development of new knowledge and techniques and the discarding of the old are still rapid in computer science and technology, at least in relation to other scientific and technical disciplines. But the pace is no longer so breakneck as it was in the 1950s and 1960s when a computer system became obsolete every two or three years; the effective and useful life of an encyclopedia of computer science today, like that of a computer system, should be measured in terms of half-decades or more. Moreover, while parts of any encyclopedia become obsolete after a time, this one contains a major proportion of material which will continue to be of reference value for many years to come.

Five years ago the scope of, as well as the need for, this encyclopedia seemed sufficiently clear that I believed this volume could be efficiently and expeditiously developed. How naive I was! Despite a long and relatively broad association with book publishing, I underestimated the scientific, administrative, and production complexities of a project as large as this one. But, if the result has taken longer to achieve than I anticipated, I do not regret the effort. Editing an encyclopedia like this one is an education itself in one's own discipline. And I value considerably the contacts with the members of the Editorial Board, all eminent computer scientists whose advice has much improved the quality of this volume, and with the over 200 authors of articles, all of whom I love, even the most recalcitrant and prima donnaish of them!

Anthony Ralston

February, 1976

Note

The Editor and Publisher would appreciate an indication from readers of how future editions of this Encyclopedia could be improved. What additional subjects need to be covered? Which articles need improvement? Any such comments or notification of errors found should be sent to Petrocelli/Charter, 641 Lexington Avenue, New York, New York 10022.

EDITOR'S FOREWORD

An encyclopedia has one main purpose—to be a reference work for the layman or the non-specialist who needs elaboration of a subject in which he is not expert. The implication of “basic” is that an encyclopedia, while it should attempt to be comprehensive in *breadth* of coverage, cannot be comprehensive in the *depth* with which it treats most topics. An encyclopedia should, however (and this one does), direct the reader to information at the next level of depth through cross-references to other articles and bibliographic references.

What constitutes breadth of coverage is always a difficult question and especially so for computer science. As a new discipline that has evolved over the past three decades, and which is still changing rather rapidly, its boundaries are blurred. This is complicated further because there is no general agreement among computer scientists or technologists about whether certain areas are or are not part of computer science.

The choice of specific subject matter for this encyclopedia has been necessarily a personal one by the Editor, modulated by the Assistant Editor, the Editorial Board, and by the practical problems of finding authors to write particular articles. My hope is that, while inevitably there will be quibbles about the inclusion of certain topics, little or nothing of major importance has been omitted.

An encyclopedia is *not* a handbook, which is normally intended only for practitioners in the subject area or for professional users of the subject area knowledge. Neither is it a *dictionary* nor a *glossary*.

Articles in this encyclopedia normally contain definitions of the article titles, but even the shortest articles also contain explanatory information to broaden and deepen the reader's understanding. Long articles contain historical and survey information in order to integrate the subject matter and put it into perspective. Overall, it is a basic reference to computer science as well as a broad picture of the discipline, its history, and its directions.

Organization

The organization of this volume is on an alphabetic basis according to the first word of each article title. Titles have been chosen in such a way that the first word is the one most likely to be selected by the reader searching for a given topic. In addition, main cross-references have been provided when more than one word in a title might reasonably be referenced. These cross-references are also used to refer to important subjects that are included in longer, more general articles rather than as separate articles.

Three additional aids to the reader have been provided. The first is the **CROSS-REFERENCES** at the beginning of each article, which list titles of other articles and names of terms used which may be unfamiliar to the reader.

The **APPENDICES** at the back of the book constitute the second aid. These include lists of abbreviations, acronyms, special notation and terminology, as well as some useful numerical tables.

The third aid is the **INDEX**. In a dictionary or glossary, all terms appear as entries, but in an encyclopedia only the most important terms are used as article titles or even main cross-references. Without an Index the location of much important information would be left to the ingenuity of the reader. In fact, the Index contains *all* terms that should appear in a *dictionary* of computer science. In addition, it contains entries that would not normally appear in a dictionary, such as references to subcategories. The encyclopedia user who searches among the article titles unsuccessfully will find

EDITOR'S FOREWORD

the Index invaluable in locating specific information. In addition, the Index will often provide pointers to unfamiliar terms.

Using the Encyclopedia

Even a rapidly developing discipline such as computer science exhibits some coherent internal structure. We have been guided in the development of this encyclopedia by our perception of this structure. Five articles cover broad disciplinary subject matter:

Computer Science
Data Processing
Information Science
Information Processing
Symbol Manipulation

The remaining articles may be grouped under ten headings:

- I. Software*
- II. Hardware*
- III. Computer Systems*
- IV. Basic Terminology*
- V. Theory*
- VI. Mathematics for Computer Science*
- VII. Applications*
- VIII. Management, Societal, Economic, and Legal Aspects*
- IX. Professional and Educational Aspects*
- X. History*

To aid the user in grasping the overall taxonomy of computer science, a **CLASSIFICATION OF ARTICLES** precedes the main body of the Encyclopedia. It includes *all* article titles, except those five designated above as broad disciplinary subject matter, as well as some additional headings. All headings that are not article titles are preceded by an asterisk (*). The **CLASSIFICATION OF ARTICLES** will enable most readers who wish to concentrate on a particular area of computer science to find a list of relevant articles. In addition, the following lists provide useful groupings of articles not adequately reflected in the **CLASSIFICATION**.

I. Basic Disciplinary Areas of Computer Science. As an academic discipline, computer science is well established and its basic content is fairly clear. The article "Education in Computing Science" overviews the subject matter of the curricula at the graduate and undergraduate levels. Topics considered to be major subdisciplines of computer science and (or) which form the subject matter of one or more college courses are listed below. As in the **CLASSIFICATION OF ARTICLES**, italicized titles are for grouping purposes only and do not refer to actual articles.

(a) *Software and Programming-Related:* Programming Languages; Language Processors; Operating Systems; Machine and Assembly Language Programming; Procedure-Oriented Languages; Data Structures; Files; Programming Linguistics; Structured Programming.

(b) *Hardware-Related:* Computer Architecture; Computer Circuitry; Logic Design; Microprogramming.

(c) *Computer Systems:* Computer Networks; Information Systems; Management Information Systems; Time Sharing.

EDITOR'S FOREWORD

(d) *Theory*: Algorithms, Analysis of; Algorithms, Theory of; Computational Complexity; Formal Languages.

(e) *Mathematics of Computer Science*: Automata Theory; Numerical Analysis; Sequential Machines.

(f) *Applications*: Artificial Intelligence; Computer-Assisted Learning and Teaching; Computer Graphics; Image and Picture Processing; Information Retrieval; Pattern Recognition; Simulation.

2. *Scientific Computing and Applications*. The following categories contain major articles relating specifically to the use of computers in science and technology, and to articles on applications in science and technology.

(a) *Software and Programming-Related*: Algebraic Manipulation Languages; Mathematical Software; Problem-Oriented Languages; Simulation: Languages.

(b) *Applications*: Computer-Aided Design; Computer Graphics; Control Applications; Engineering Applications; Image and Picture Processing; Medical Applications; Pattern Recognition; Scientific Applications; Simulation: Principles; Speech Recognition; Text-Editing Systems.

This list is not exhaustive, since other articles also contain topics relevant to scientific-technical applications. Conversely, most of the articles listed above contain material applicable to other areas.

3. *Administrative and Business Data Processing*: The following major articles are related specifically to the use of computers for administration and business and to articles on applications in these areas.

(a) *Software and Programming-Related*: Access Methods; Data Base and Data Base Management; Data Security; Decision Table Languages; Files; Nonprocedural Languages; Software Packages.

(b) *Applications*: Administrative-Business Applications; Credit Applications; Information Systems; Management Information Systems; Planning Applications; Sorting.

Important aspects of these articles are relevant beyond administrative and business data processing.

The foregoing lists and the Classification of Articles that follows have been especially designed to guide curriculum development, to satisfy the requirements of the computer specialist outside his/her areas of expertise, to direct the readings of lay persons who may wish to become familiar with particular aspects of computer science, or to guide readers in following a self-study regime.

It would be pretentious to claim that the Encyclopedia will be "all things to all people," but I am confident that it will fill a much-needed basic reference in the field of computer science.

ANTHONY RALSTON

CLASSIFICATION OF ARTICLES

(Note: This classification list includes all article titles except the five designated as broad disciplinary subjects: All headings that are not article titles are preceded by an asterisk.)

I SOFTWARE

PROGRAMMING LANGUAGES

- Algebraic Manipulation Languages
- Associative Languages
- Authoring Languages and Systems
- Command and Job Control Languages
- Decision Tables: Languages
- List-Processing Languages
 - Garbage Collection
- Macrolanguages
- Nonprocedural Languages
- Problem-Oriented Languages
- Procedure-Oriented Languages: Survey of
 - Algol 68
 - Extensible Language
 - Pascal
- Simulation: Languages
- String Processing Languages

*SYSTEMS SOFTWARE

- Assemblers
- Input-Output Control Systems
- Interpreter
- Language Processors
 - Arithmetic Scan
 - Binding Time
 - Compatibility
 - Compile and Run Time
 - Compiler, Incremental
 - Compiler, Syntax-Directed
 - Load-and-Go Compiler
 - Reentrant Program
 - Side Effect
- Macroinstruction
- Operating Systems
 - Bootstrap
 - Buffer
 - Deadlock
 - Linkage Editor
 - Loader
 - Nucleus
 - Overhead

CLASSIFICATION OF ARTICLES

- Semaphore
- Spooling
- Supervisor Call
- System Generation
- Working Set
- Software Packages
- Utility Program

*PROGRAMMING

- Machine and Assembly Language Programming
 - Breakpoint
 - Dump
 - Loop
 - Patch
 - Systems Programming
- Procedure-Oriented Languages: Programming
 - Applications Programming
 - Backtracking
 - Character Set
 - Checkpoint and Restart
 - Concatenation
 - Controlled Variable
 - Coroutine
 - Dangling ELSE
 - Delimiter
 - Object Program
 - Overlay
 - Source Program
- Structured Programming
 - Modular Programming

*PROGRAM AND DATA STRUCTURES

- Block Structure
- Constants
- Data Structures
 - Data Type
 - FIFO-LIFO
 - Pointer
 - Ring
 - Stack
 - Tree
- Data Structures, Set Concepts for Files
 - Binary Search
 - Catalog
 - Collating Sequence
 - Data Set
 - Hashing
 - Key
 - Open and Close a File

CLASSIFICATION OF ARTICLES

- Record
- Update
- Procedure
 - Argument
 - Global and Local Variables
 - Procedure, Pure
- Statements
 - Declarative Statement
 - Executable Statement
- Subprograms, Calling

PROGRAMMING LINGUISTICS

- Grammar, Generative
- Grammar, Reductive
- Parsing
 - Precedence
 - Production
- Syntax, Semantics, and Pragmatics

SOFTWARE ENGINEERING

- Portability
- Software Flexibility
- Software Maintenance

*LIBRARIES

- Mathematical Software
- Program Libraries

II *HARDWARE

*COMPUTERS

- Analog Computers
- Differential Analyzer
- Digital Computers: General Principles
 - Calculators, Desk
 - Calculators, Electronic
 - von Neumann Machine
- Hybrid Computers
- Minicomputers
 - Microcomputer
- Special-Purpose Computers
 - Data Acquisition Computer
- Supercomputers

*MEMORY AND PERIPHERALS

- Addressing
 - Address Modification
 - Computers, Multiple Address
 - Indirect Address

CLASSIFICATION OF ARTICLES

- Channel
- Communication Control Unit
- Data Preparation Devices
 - IBM Card
 - Ninety-Column Card
- Digital-to-Analog Converters
- Input-Output Devices
 - Card Reading and Punching Techniques
 - Hard Copy
 - Keyboard Standards
 - Machine-Readable Form
 - Paper Tape
 - Printing Techniques
- Memory: Main
 - Associative Memory
 - Base Register
 - Cache Memory
 - Contention
 - Cycle Stealing
 - Cycle Time
 - Interleave
 - Interlock
 - Lockout
 - Memory Protection
 - Ports, Memory
 - Thrashing
 - Ultrasonic Memory
 - Williams' Tube Memory
- Memory: Auxiliary
 - Access Time
 - Block and Blocking
 - Cyclic Redundancy Check
 - Cylinder
 - Direct Access
 - Latency
 - Logical and Physical Units
 - Original Equipment Manufacturer (OEM)
 - Parity
 - Scratch File
 - Tape Label
- Multiplexing
- Optical Character Readers
 - Optical Mark Readers
- Storage Allocation
- Storage Hierarchy
- Storage Organization
 - Word Length, Variable
- Terminals
 - Audio Response Terminal

CLASSIFICATION OF ARTICLES

- Intelligent Terminal
- Point-of-Sale Terminal

CENTRAL PROCESSING UNIT (CPU)

- Arithmetic-Logic Unit
 - Adder
 - General Register
 - Index Register
 - Register
- Computer Circuitry
 - Integrated Circuitry
- Interrupt
 - Program Status Words and State Vectors
- Logic Design
- Machine Instruction Set
 - Decrement
 - Input-Output Instructions
 - Operand
 - Operation Code
 - Privileged Instruction
 - Shifting
- Main Frame
- Program Counter

PERFORMANCE OF COMPUTERS

- Benchmark
- Grosch's Law
- Hardware Monitor
- Maintenance of Computers
- Reliability and Fault Tolerance
 - Redundancy
- Throughput
- Turnaround Time

COMPUTER ARCHITECTURE

- PMS Notation

III COMPUTER SYSTEMS

COMPUTER NETWORKS

- ARPA Network
 - Interface Message Processor (IMP)
- Communications and Computers
- Computer Utility
- Data Communications
 - Acoustic Coupler
 - Bandwidth
 - Baud

CLASSIFICATION OF ARTICLES

- Conditioning
- Data Communication Networks
- Handshaking
- Modem
- Noise
- Packet Switching
- Networks for Instruction
- Teleprocessing Systems
- Front End

PROCESSING MODES

- Multiprogramming
- Multiprocessing
- Open and Closed Shop
- Parallel Processing
- Remote Job Entry (RJE)

TIME SHARING

- Scheduling Algorithm
- Swapping
- Time Slice

COMPUTER, USING A

- Computing Center
- Debugging
 - Trace
 - Trap
- Diagnostics

*STORAGE MANAGEMENT

- Access Methods
- Data Base and Data Base Management
- Data Security
- Storage Management Structures
- Virtual Memory
- Volume

* SYSTEM MANAGEMENT

- Computer Accounting and Resource Control
- Performance Measurement and Evaluation

MICROPROGRAMMING

- Control Point
- Emulation
- Host System
- Local Store
- Read-Only Store

IV “BASIC TERMINOLOGY

*PROGRAMMING-RELATED

- Algorithm
 - Markov Algorithm
 - Parallel Algorithm
- Flowchart
 - Block Diagram
 - Flow Diagram
 - System Chart
- Heuristics
- Identifier
- Iteration
- Job
- Label
- Lists and List Processing
 - String
- Masking
- Program
 - Subroutine
- Recursion
- Stored Program Concept
- Task

*GENERAL

- Automation
- Cybernetics
- Errors
 - Errors, Absolute and Relative

*Jargon

- Bug
- Fix
- GIGO
- Glitch
- Kludge
- Ping-Pong

Models

V *THEORY

- Algorithms, Analysis of
- Algorithms, Theory of
- Computability
- Computational Complexity
- Decidability
- Formal Languages
 - Backus-Naur Form
 - Meta Character

CLASSIFICATION OF ARTICLES

- Meta Language
- Meta Variable
- Regular Expression
- Vienna Definition Language
- Well-Formed Formula
- Information and Data
- Programming Correctness, Proof of
- Programming Language Models

VI *MATHEMATICS FOR COMPUTER SCIENCE

*NUMERICAL MATHEMATICS

- Approximation Theory
 - Chebyshev Approximation
 - Least Squares Approximation
- Arithmetic, Computer
 - Precision
 - Significance Arithmetic
- Complements
- Error Analysis
- Fast Fourier Transform
- Interval Arithmetic
- Matrix Computations
- Numbers and Number Systems
 - Significant Digit
- Numerical Analysis
 - Finite Element Method
 - Partial Differential Equations, Numerical Solution of
- Roundoff Error
- Table Lookup

*ALGEBRA AND AUTOMATA THEORY

- Automata Theory
 - Cellular Automata
 - Probabilistic Automata
- Boolean Algebra
 - Polish Notation
- Sequential Machines
- Turing Machines

*STATISTICS AND RELATED TOPICS

- Decision Tables: General Principles
- Queueing Theory
- Random Number Generation
- Monte Carlo Method
- Regression Analysis
- Stochastic Process

CLASSIFICATION OF ARTICLES

CODES

- ASCII
- Baudot Code
- Binary-Coded Decimal, Natural
- EBCDIC
- Error-Correcting Codes

OPERATIONS RESEARCH

- Mathematical Programming
- Simplex Method

GRAPH THEORY

LAMBDA CALCULUS

VII *APPLICATIONS

*COMPUTER SCIENCE APPLICATIONS

- Artificial Intelligence
 - Theorem Proving
- Computer Graphics
 - Cursor
 - Joystick
 - Lightpen
 - Pictures, Basic Structure
 - Rand Tablet
- Image and Picture Processing
- Information Retrieval
 - Current Awareness Systems
 - Keyword-in-Context (KWIC) Index
 - Medlars-Medline
- Pattern Recognition
 - Perceptron
- Sorting
 - Sort-Merge Packages

*OTHER APPLICATIONS

- Administrative-Business Applications
 - Exception Reporting
 - Management Information Systems
- Arts Applications
- Computer-Aided Design
- Control Applications
- Credit System Applications
- Cryptography, Computers in
- Economic Applications
- Engineering Applications

CLASSIFICATION OF ARTICLES

Games on Computers
Humanities Applications
Information Systems
 Hospital Information Systems
Language Translation
Medical Applications
 Biomedicine, Computer Graphics in
 Intensive Care, Computers in
Planning, Computer Applications in
 PERT/CPM
Real-Time Applications
Scientific Applications
Simulation: Principles
Social Science Applications
Speech Recognition
Statistical Applications
Text Editing Systems
 Justification

VIII *MANAGEMENT, SOCIETAL, ECONOMIC, AND LEGAL ASPECTS

COMPUTERS AND SOCIETY

Data Bank

COMPUTING ECONOMICS: ACQUISITION AND OPERATION

Unbundling

SERVICE BUREAUS, DATA PROCESSING

Turnkey

COPYRIGHTS AND PATENTS, COMPUTER ASPECTS OF

Legal Protection of Software

Proprietary Program

CRIME AND COMPUTER SECURITY

SECURITY OF COMPUTER INSTALLATIONS, PHYSICAL

DOCUMENTATION

Feasibility Study

PERSONNEL IN THE COMPUTER FIELD

PLANNING SYSTEMS, CHARACTERISTICS OF

SOFTWARE MANAGEMENT

STANDARDS

IX *PROFESSIONAL AND EDUCATIONAL ASPECTS

COMPUTER-ASSISTED LEARNING AND TEACHING

Computer-Assisted Instruction (CAI)
Computer-Managed Instruction (CMI)

EDUCATION IN COMPUTING SCIENCE

HIGHER EDUCATION, COMPUTERS IN

LITERATURE IN COMPUTING

COMPUTER USER GROUPS

CUBE
GUIDE
Joint Users Group (JUG)
SHARE
USE
VIM

***COMPUTER SOCIETIES**

American Federation of Information Processing Societies (AFIPS)
American Society for Information Science (ASIS)
Association for Educational Data Systems (AEDS)
Association for Computing Machinery (ACM)
Association Française pour la Cybernetique, Economique et
Technique (AFCET)
British Computer Society (BCS)
Conference on Data Systems Languages (CODASYL)
Data Processing Management Association (DPMA)
Institute for Certification of Computer Professionals (ICCP)
Institute of Electrical and Electronic Engineers-Computer
Society (IEEE-CS)
Intergovernmental Bureau for Informatics (IBI)
International Federation of Automatic Control (IFAC)
International Association for Analog Computation
International Federation for Information Processing (IFIP)
Society for Computer Simulation (SCS)
Society for Industrial and Applied Mathematics (SIAM)

X *HISTORY

DIGITAL COMPUTERS: EARLY, and ORIGINS

*Computers
Atlas
EDSAC
EDVAC
ENIAC

CLASSIFICATION OF ARTICLES

Livermore Automatic Research Computer (LARC)
MARK I
SEAC
Stretch
SWAC
UNIVAC I
Whirlwind
Generations, Computer

*PEOPLE

Aiken, Howard
Atanasoff, John
Babbage, Charles
Boole, George
Eckert, J. Presper
Eckert, Wallace
Hollerith, Herman
Leibniz, Gottfried Wilhelm von
Mauchly, John
Pascal, Blaise
Turing, Alan
von Neumann, John
Watson, Thomas, Sr.
Wiener, Norbert
Wilkes, Maurice V.
Zuse, Konrad

MANUFACTURERS, COMPUTER

Control Data Corporation 6000 Series
IBM 1400 Series
IBM 360-370 Series
RAMAC



ACM. See ASSOCIATION FOR COMPUTING MACHINERY.

ASIS. See AMERICAN SOCIETY FOR INFORMATION SCIENCE.

AEDS. See ASSOCIATION FOR EDUCATIONAL DATA SYSTEMS.

AFCET. See ASSOCIATION FRANÇAISE POUR LA CYBERNETIQUE, ECONOMIQUE ET TECHNIQUE.

AFIPS. See AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES.

APL. See PROCEDURE-ORIENTED LANGUAGES.

APT. See PROBLEM-ORIENTED LANGUAGES.

ACCESS METHODS

For articles on related subjects see **DATA BASE AND DATA BASE MANAGEMENT; FILES; HASHING; KEY; and RECORD.**

An access method is a technique of accessing data that has been placed in some kind of storage. This term was derived from the earlier term "access time," which concerned the speed with which data could be located on a storage device. The use of the term "access method" became popular in about 1964-1965 when it was used by IBM in connection with OS/360. Although the term implies a way of getting to data that is already stored, it is primarily a way of storing data under circumstances that also dictate the way it is to be subsequently accessed.

In a typical situation a choice of such access methods is available, and one method must normally be chosen for each file. For the purposes of this article, a *file* is defined as a collection of records usually, but not necessarily, all of the same type (hence, the occasional term "file access method"). An access method is therefore a property of a file whose essential purpose is to instruct how the file is to be stored. There may then be several ways of accessing or retrieving the records in the file.

ACCESS METHODS

Classes of Access Methods. If a file is stored on a device such as magnetic tape, then the only type of access available is pure sequential access, in which a search is made starting from the beginning of the file. Thus, the concept of an access method really became of interest only with the advent of direct access devices such as disks and drums. On such devices, data may be accessed directly by the address, or location, of the data on the device. Direct access of this kind is rare because the precise location of a data record in a storage device is seldom known to the software system that desires the access. It is necessary, therefore, to have available indirect methods for accessing data.

Indirect methods may be classified (1) as sequential, in which there is some type of search through a sequence of records (but generally not a complete search that starts from the first record and proceeds through the whole file), or (2) as **nonsequential**, in which the desired record is located without such a search. A common nonsequential method is based on the use of an index and is usually called "Index Sequential Access Method" (**ISAM**). Other common nonsequential access methods use key transformation techniques and are usually referred to as "randomizing" or "hashing" techniques.

Index Sequential Access Method. If an index is used, then some extra storage space is required to hold the index. In **ISAM**, one or more items in the record type are chosen as the **ISAM** "key." If the file has in fact two or more record types, then the **ISAM** key must be an item or items to be found in all record types.

The index then consists of an ordered sequence of the values of the **ISAM** key in the collection of records that are initially stored using this access method. Associated with each value is a list of addresses or pointers to the records, which have this value as their prime key. As an example, consider a file of records, each containing data about an employee in a company. Each employee has an employee number that distinguishes him from all other employees in the company. In other words, no two employees have the same employee number. One item in each record type would be the employee number and if this file of employee data is to be stored as an **ISAM** file, this item would most probably be chosen as the **ISAM** prime key. This means that it would always be possible to retrieve an employee record from the file if the employee number is known. As indicated above, the file is assumed to be stored on some kind of direct access storage such as disk or drum.

Typically with an **ISAM** file indexed on a single key in this way, the index could be used for updating. In other words, if the file contains 10,000 records and it is necessary to update 25 of these, then the index can be used to access the 25 records separately without unnecessarily accessing the other 9,975, as would be necessary if the file were stored on tape or if it were stored on disk without an index being available for use. If a new record is added to the file, then the **ISAM** index is automatically updated by the indexing mechanism to include the new employee number and the address of the employee record containing it in the file.

In view of the fact that **ISAM** is often used for very long files containing possibly hundreds of thousands of records, the index table itself may become quite long. In the most simple case, it would be necessary first to scan this long index table for a value when it was necessary to find a record having this value. One way of alleviating this problem is to associate with a group of key values a single address or pointer to the beginning of a block of records which contains the records with these key values. This block would then be searched either sequentially or by binary search to find the record with the desired key. If the blocks are not large, this does not introduce serious inefficiency. Moreover, this scheme has the advantage of facilitating insertions into, and deletions from, the file if the blocks are set up initially with extra locations.

If, however, the blocks are small, the index table may still be quite long. Therefore, the idea of a hierarchy of indexes (Table 1) has evolved to **expe-**

Table 1 . A Hierarchical Block-Oriented Index.			
Index Level 1	Index Level 2	Block Number	Block Starting Address
00000-08756	00000-00713	1	46217
	00718-01426	2	46337
	.	.	.
	07823-08756	10	47395
41063-52071	41063-42217	41	50362
	.	.	.
	49278150593	49	51612
	506 14-5207 1	50	51738

dite the process of finding a value in an index table. For example, suppose an employee file with 10,000 records has employee numbers in the range 00000–99999. To retrieve the record for employee number 49,731, the first level would be searched until the fifth entry (410635207 1) was found, and then the second level would be searched until the ninth entry (49278-50593) was found, which would be substantially more efficient than searching through 49 entries of a single level index. The key 49731 is in block 49, with a starting address (usually on a disk) of 5 16 12. The addresses starting with 5 1612 would then be searched sequentially for the record with key 4973 1. While the 100 blocks hold an average of 100 employee records each, the blocks generally contain more than 100 locations. This allows change and growth of the file before it must be reorganized and the index table updated.

There are many different approaches to using indexes in file access, as well as other access methods that do not involve indexes. Acronyms for some of these (e.g., VSAM, Virtual Sequential Access Method; QSAM, Queued Sequential Access Method) have arisen from IBM operating system terminology.

Access Methods Using Key Transformation. In this class of access methods, no index is maintained as in the other class. Instead, to decide where in storage to place the record, the value of the key is used as input to some algorithm that is designed to produce as output an address in storage. This address is not normally a physical address, in the sense of stipulating exactly which physical location in direct access storage will be used. It is usually a logical address within some area, or extent, or realm (all these words are in common use).

An example of the simplest algorithm is the following: Assume the space in which the records are to be stored is divided into N equal sized “pages,” each of which can store some number M of records. It is necessary for N to be a prime number for the algorithm to be effective. If the value of the chosen key is divided by N , then the result will be

$$\frac{\text{Key value}}{N} = I + R$$

where I is the integral part of the quotient and R is the remainder. Then R identifies the page in which the record is to be stored.

When more than M key values randomize (or hash) to the same value, which is a possibility with almost all such algorithms, then provision must be made to store the additional values in some kind of

overflow area (perhaps on a different storage medium). When the record is to be retrieved from storage, the same algorithm is used to obtain the page from the key value. Locating the record in the page may then be by sequential or binary search or by a further indexing or key transformation technique, depending upon the number of records stored in each page.

The fact that these algorithms distribute the records somewhat randomly throughout the available space accounts for the fact that the approach is sometimes called a “random access” method. The fact that the bits in the key value are sometimes jumbled up, or “hashed,” by the algorithm in order to produce the address accounts for the term “hashing.”

Application to Data Base Management. In many present-day host-language data base systems the term “access method” is simply not used. The reason for this is that the records are stored in the data base and there are often many different ways of accessing each. The term “access method,” as indicated, implies both a way of storage and a way of retrieval. There can, of course, be only one way in which any given record type is stored, and this has recently been called its “location mode.” In view of the unfortunate ambiguity of the word “locate,” which means both to find and to place, the term “storage mode” would have been a better choice.

. REFERENCES

- 1971. Lefkowitz, D. *File Structures for On-Line Systems*. New York: Spartan-Macmillan.
- 1971. Davis, G. B. *Introduction to Electronic Computers*. New York: McGraw-Hill.

T. W. OLLE

ACCESS TIME

For articles on related subjects see **DIRECT ACCESS**; **LATENCY**; and **MEMORY**: Auxiliary.

Direct access devices require varying times to position a read/write head over a particular record. In the case of a moving-head disk drive, this involves positioning the “comb” (head assembly, as in Fig. 1)

ACCESS TIME

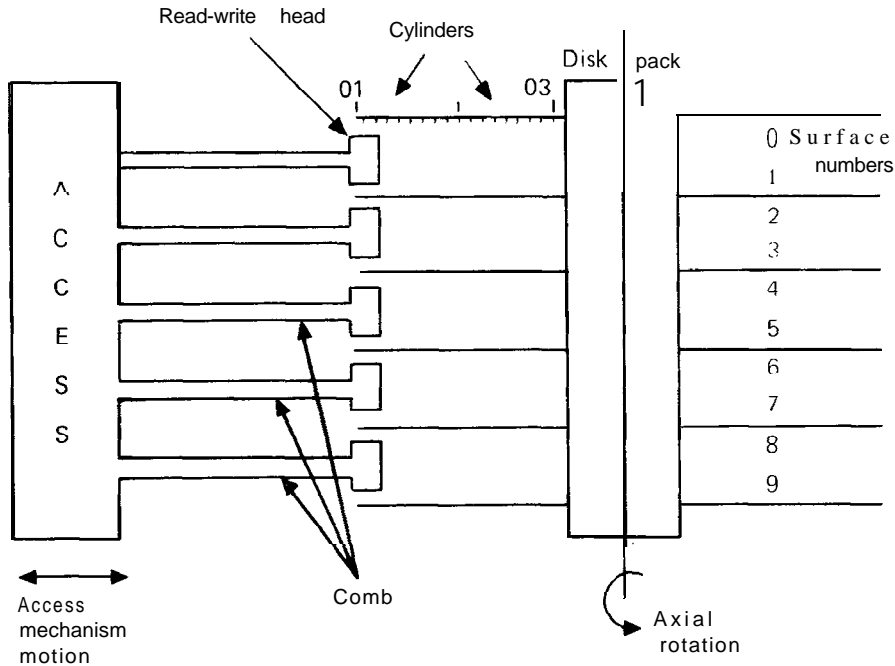


Fig. 1. Side view of typical disk drive.

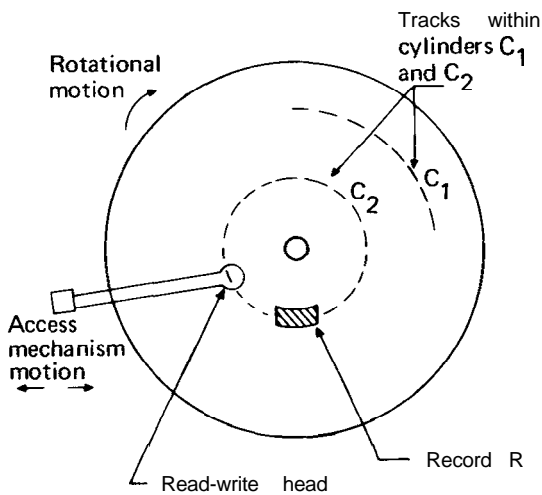


Fig. 2. Top view of typical disk drive.

to the designated cylinder, plus rotation of the selected track to the desired record. Comb-movement times for a typical medium-sized disk drive are shown in Fig. 3.

"Access time" is the sum of comb-movement and rotational times to reach a particular record.

There is a different access time for each record retrieved at random from a disk drive, since it is necessary to move from cylinder C_1 to cylinder C_2 (Fig. 2), then await rotational positioning of record R . Generally of interest are maximum *access time* for a particular device (135 ms for the disk drive of Fig. 2), average access time (60 ms for this particular drive), and *minimum access time* (25 ms—the time to move the comb to an adjacent cylinder). The latter is also called "track-to-track access time."

Average access time is an important parameter for analytical planning of a real-time computer application, e.g., an on-line inquiry system. Minimum access time is more important for sequential usage of disk drives. The dominant component of delay for sequential retrieval of records from a disk drive is the average time for a half-rotation (12.5 ms for the drive described in Figs. 1 and 2).

During the past 15 years, rotational speeds for disk drives have improved very little: 2400 rpm is typical, equivalent to 25 ms per rotation. Bit densities per track have increased fivefold in this same period, so that average transfer speeds have increased even if track-to-track access times have not diminished. During this period, average access times have been halved, as a result of a widespread changeover from hydraulic actuators to "voice coil"

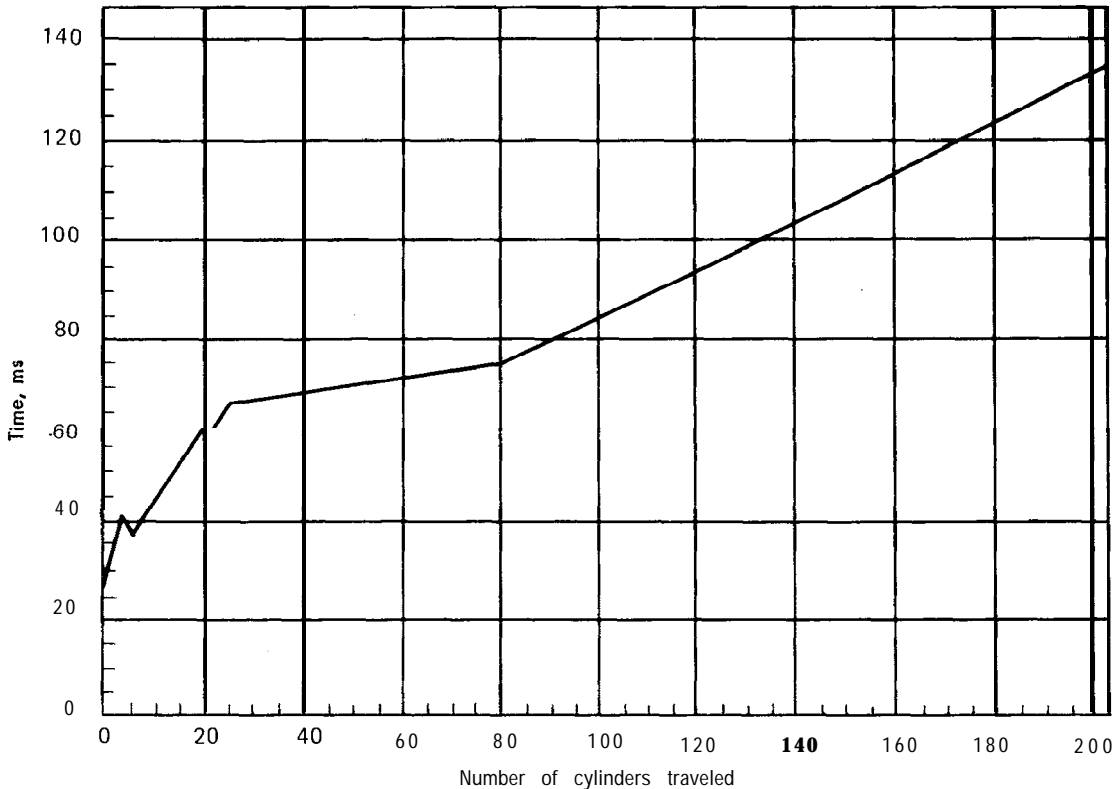


Fig. 3. Comb-movement times for typical disk drive.

actuators for moving the comb mechanism.

For a drum, average access time is a half-revolution and maximum access time is a full revolution, since drum heads are typically fixed over the data areas. Average access times for drums are 5 to 10 ms.

For magnetic card and similar mass storage systems, average access times depend on movement of the medium to a read/write head. Typical average access times for magnetic card devices are 1 sec and maximum access times are 2 to 4 sec. For tape-cartridge mass storage systems, average access time is approximately 15 sec and minimum access time to a new cartridge is approximately 12 sec.

D. N. FREEMAN

ACOUSTIC COUPLER

For article on related subject see **MODEM**.

An acoustic coupler (see Fig. 1) is a modem in which the coupling between a computer processor or terminal device and the communications line (almost always the telephone network) is acoustic rather than electric. The output of an acoustic coupler is an audible sound, which is applied directly to the telephone mouthpiece; in the reverse direction, the telephone earpiece is applied to a microphone in the modem. The advantage of an acoustically coupled modem is that almost any telephone handset can be used, and the modem is truly portable. The disadvantage is that the acoustic coupling can be noisy and may limit the speed of operation of the device; acoustically coupled modems are seldom used at data speeds above 300 bps (bits per second). With electric coupling, speeds of 1,200 bps or even 4,800 bps can be achieved with low error rates on switched telephone lines. The limiting component is usually

ACCOUNTING, COMPUTER. See **COM-PUTER ACCOUNTING AND RESOURCE CONTROL.**

ADDER

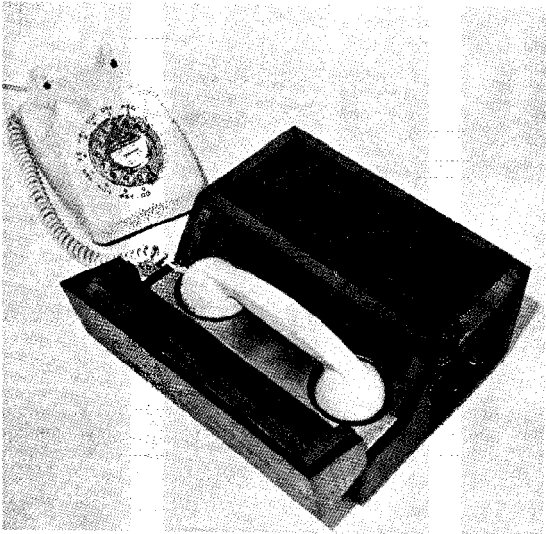


Fig. 1. Acoustic coupled modem with telephone handset. (Reproduced by kind permission of the manufacturers, K & N Electronics, Ltd., Maidenhead Berks, England.)

the microphone in the telephone handset. If this is replaced by a better one (an easy exchange, but usually not permitted by telephone companies) higher speeds could be obtained reliably.

P. T. KIRSTEIN

ADDER

For articles on related subjects see **ARITHMETIC**, **COMPUTER**; and **ARITHMETIC-LOGIC UNIT**.

The adder is a logic circuit that forms the sum of two or more numbers represented in digital form.

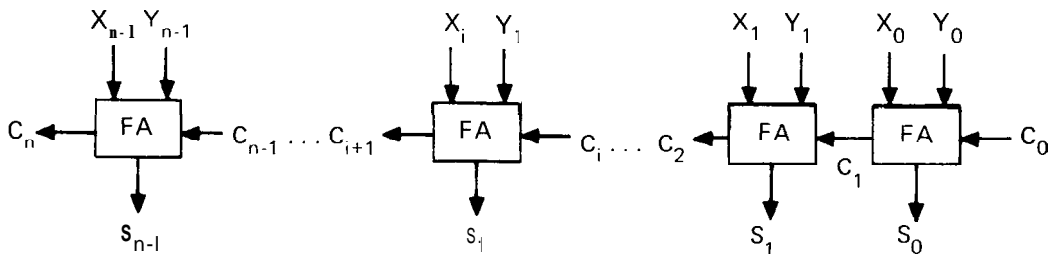
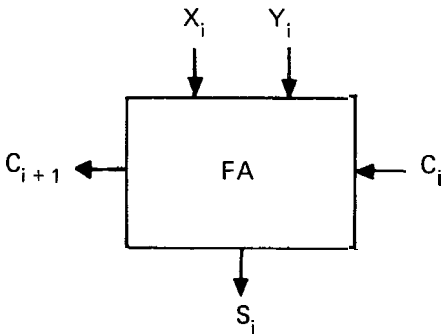


Fig. 2. Binary ripple-carry adder.

The simplest adder is the binary one-position adder, also called a “full adder” (Fig. 1). A “ripple-carry” adder for two n -bit binary numbers is formed by connecting n full adders in cascade (Fig. 2). The addition time of the ripple-carry adder corresponds to the worst-case delay, which is n times the time required to form the C_{i+1} (carry) output by one full



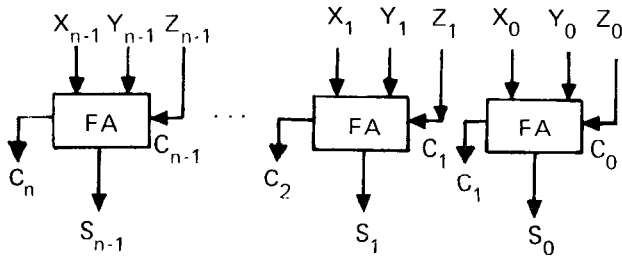
s: sum
C: Carry
(a) Diagram

X_i	Y_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

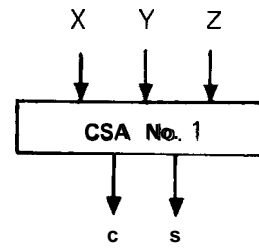
(b) Truth table

Fig. 1. The binary full adder (FA).

ADDER



(a) Detailed diagram



(b) Compact notation

Fig. 3. Three-operand binary carry-save adder,

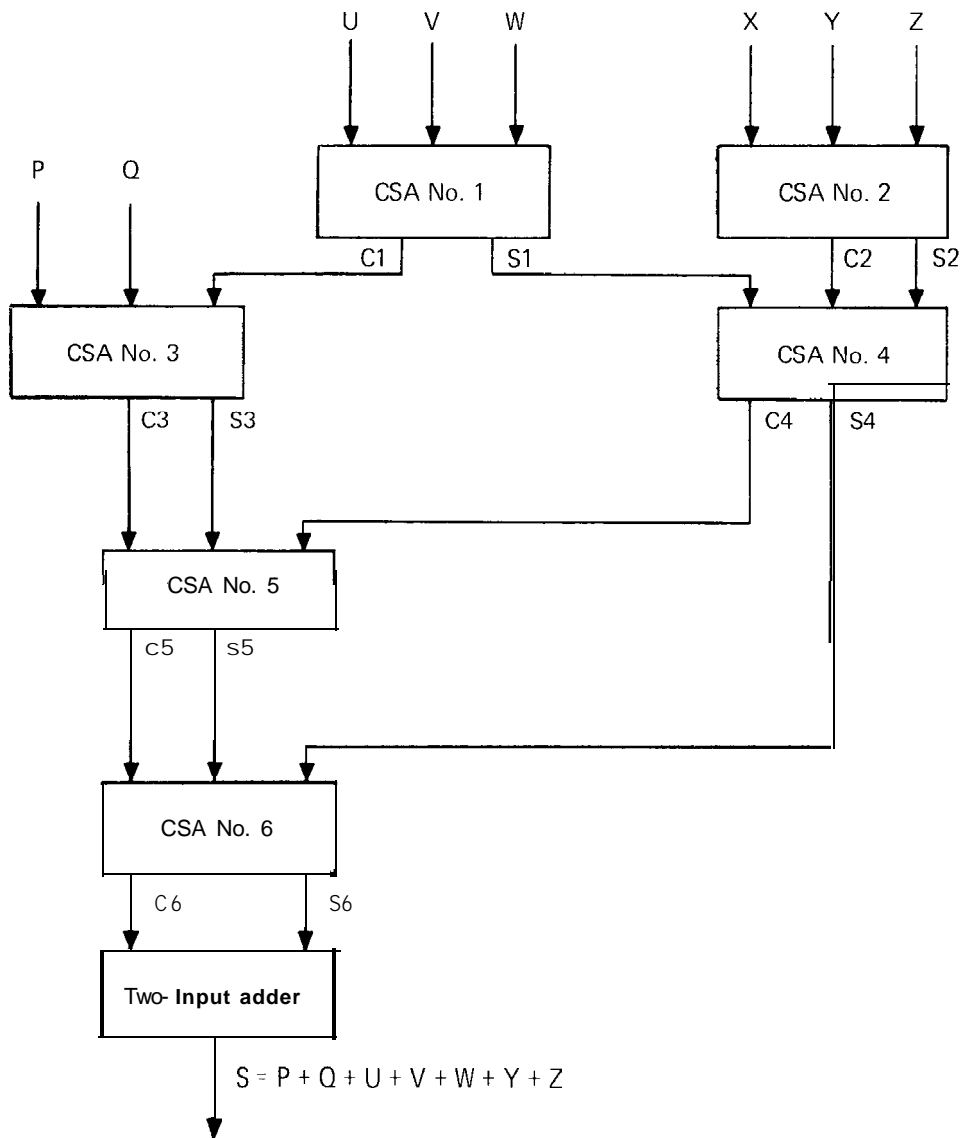


Fig. 4. CSA summation of eight operands,

ADDRESS, INDIRECT

adder, plus the time to form the S_i output, given the C_i input.

Higher speeds of two-operand addition can be attained by the use of carry-completion sensing, carry-lookahead, and conditional-sum techniques (Garner, 1965). In these techniques, additional logic circuitry is employed to reduce the total delay in the adder circuits.

One-position adders for a higher radix r (for example, 4, 8, 10, or 16) are similar to the full adder of Fig. 1. The digits X_i and Y_i assume values 0 to $r - 1$, and they are represented by two or more logic variables. The truth table describes the addition table for radix r ; carry signals (C_i and C_{i+1}) remain two-valued (0 and 1). The adder speed-up techniques discussed for radix 2 also apply to two-operand addition of higher radix numbers.

Fast summation of three or more operands can be accomplished by the use of "carry-save" adders (CSA). A binary three-operand n -bit CSA is shown in Fig. 3. The third n -bit operand Z is entered on the C_i inputs of n binary full adders. The C_{i+1} outputs form a second output word $C = (C_n \dots C_1)$ and the sum of the three input words X, Y, Z is represented by two output words C and $S = (S_{n+1} \dots S_0)$. The time required to form C and S is equal to the time required by one binary full adder. The final sum, which is the sum of C and S , is then obtained in a two-operand adder, which may employ any of the speed-up techniques discussed above.

The summation of more than three operands uses CSAs in a similar manner to reduce the sum to two words. Fig. 4 illustrates the CSA configuration for eight operands P, Q, U, V, W, X, Y, Z . The abbreviated notation of Fig. 3(b) is employed. The time required to form the words C_6 and S_6 (representing the sum of the eight input operands) is equal to four full-adder operation times, regardless of the length of the operands.

Carry-save adders are frequently employed to implement fast multiplication by means of multiple-operand summation. The technique of "pipelining" may be employed to further improve the effective speed of CSA utilization.

REFERENCE

1965. Garner, H. L. "Number Systems and Arithmetic," in F. Alt and M. Rubinoff (Eds.), *Advances in Computers*, Vol. 6, pp. 131-194. New York: Academic Press.

ADDRESS, INDIRECT. See **INDIRECT ADDRESS**.

ADDRESS MODIFICATION

For article on related subject see **ADDRESSING**.

For articles on related terms see **INDEX REGISTER**; and **REENTRANT PROGRAM**.

The idea that programs are data and can be stored just as data is stored in an electronic memory device was one of the most important ideas in the development of the stored program computer. When programs are stored as data, it is possible to bring instructions into the arithmetic unit of the computer and perform all arithmetic and logical operations of the computer on the instructions themselves.

This was of utmost importance on many of the early computers. A great deal of the power of the computer comes from its ability to execute loops in which the same program segment is applied to an array or sequence of data elements. In most of the early stored program computers, the only way of doing this was by successively changing the addresses of the instructions in the loop.

Fig. 1 is a program that adds 100 numbers stored in locations 2001 to 2100. The program itself is in locations 242 through 259 and the answer is placed in 241. This program is typical of the execution of loops through address modification on many early computers.

Even for this very simple calculation, the main loop requires the execution of nine instructions in order to add just one more number to the sum.

The use of index registers eliminated much of this inefficiency, but address and instruction modification is still used in many programming situations.

Many recent computing systems use reentrant code for all or some of the programs executed. Reentrant code cannot be modified during the execution of a program, and techniques such as address modification cannot be used in conjunction with it.

A. AVIŽIENIS

S. ROSEN

	241	(Location of ultimate result)
	242 FETCH 259 }	Initialize sum to zero (242 begins program)
	243 STORE 241 }	
	244 FETCH 256 }	Initialize location 250
	245 STORE 250 }	
Main Loop	246 FETCH 250 }	
	247 ADD 257 }	Modify address in 250 to add next number
	248 STORE 250 }	
	249 FETCH 241 }	
	250 [ADD 2000 + n] }	Do the addition (Instruction in location 250 is the one modified; the contents of this location before execution of the program are irrelevant. It is initialized by the instruction in location 245.)
	251 STORE 241 }	
	252 FETCH 250 }	
	253 SUBTRACT 258 }	
	254 BRANCH ON NEG 246 }	Check if all numbers have been added
	255 JUMP OUT }	
	256 ADD 2000 }	Constants used by the program.
	257 000 000 001 }	Note: some of the constants look
	258 ADD 2100 }	like instructions.
	259 000 000 000 }	

Fig. 1. Use of address modification in a simple program on an early computer.

ADDRESSING

For articles on related subjects see **MA-CHINE INSTRUCTION SET**; **STORAGE ALLO-CATION**; and **STORAGE ORGANIZATION**. For articles on related terms see **BASE REGISTER**; **GENERAL REGISTER**; **INDEX REG-ISTER**; **INDIRECT ADDRESS**; **OVERLAY**; and **VIRTUAL MEMORY**.

BASIC TERMINOLOGY AND HARDWARE CONCEPTS

A typical computer instruction must indicate not only the operation to be performed but also the location of one or more operands, the location where the result of the computation is to be deposited, and, sometimes, the location where the next instruction is to be found. Of course, in certain kinds of instructions such as those involved in decision making, there may be no computational operands but only a determination of the next instruction to be executed. Normally, however, all parts of the instruction are either explicitly or implicitly given. We will first consider the hardware techniques by which an address (or location) in the computer may be specified. In what follows, we shall consider primarily storage in which each location has associated with it a sequentially assigned address. An alternative meth-

od of determining a desired storage location will be considered briefly in the later section "Content-Addressable Storage."

Historically and presently, computer hardware allows addresses to be specified in a variety of ways. The most straightforward approach would be to put the entire address directly into the instruction, representing a specific location of a word or part of a word in storage. Thus, on the IBM 650, an early decimal computer, the 2-digit operation code, and the two 4-digit addresses, representing the location of the data and the location of the next instruction, respectively, were represented in the instruction itself. (It should be noted that, on modern computers, except for the case of decision-making instructions, the address of the next instruction is virtually always taken implicitly to be the location after that of the instruction being executed.) The operation code in the 650 (as on modern computers) implied the location of one of the operands and the location of the result.

Op	Data	Next Inst.
Code	Address	Address
2 digit	4 digit	4 digit

For example, the operation code **AU** (add to upper) implied that the upper half of the accumulator register was one of the operands, along with the explicitly named operand, and the result was to remain in the upper half of the accumulator.

ADDRESSING

As the amount of storage increases, however, and the number of digits (either binary or decimal) needed to represent an address becomes large relative to the size of the instruction, it becomes clear that it is no longer feasible to represent an entire address each time it occurs in an instruction. This is especially true when the address part of an instruction must be able to accommodate the largest possible storage that might be attached to a particular model of computer, even though an individual installation might only have a small part of that storage. In such cases the addresses actually occurring use only a small portion of that part of the instruction set aside for addresses. The remaining portion must always contain zeros, representing a waste of a valuable resource.

Several hardware devices have been and are employed to obtain, from one of a small number of larger registers, most of the information needed to specify an address, with the instruction itself containing only the information needed to complete the address. A number of these methods were employed in the Control Data Corp. (CDC) 160 and 160A computers, early small machines that started out with 4,096 12-bit words of storage. In the CDC 160, which dates back to 1959, six bits were used for the operation code, while the other six bits (with only 64 possible values) were used in the determination of an address. By choosing an appropriate operation code, the address would be interpreted to be in one of five modes: direct address (d); indirect address (i); relative address forward (f), and backward (b); and no address (n).

The direct addressing mode (d-mode) corresponds to the IBM 650 situation discussed above in that the address referred to a 12-bit operand in one of the first 64 words of storage.

Relative addressing provided for operand addresses and jump addresses that were near the storage location containing the current instruction. In relative addressing forward (f), the six-bit address portion was added to the current contents of the program control register P. (This register held the full 12-bit address of the current instruction.) The new value was then used for obtaining the operand, or used to jump to one of the 63 addresses forward from the address holding the instruction that was being executed. For relative addressing backward (b), the operand or jump address was obtained by subtracting the six-bit address from the current contents of I?

In the no-address mode (n), which is usually now referred to as the "immediate" mode, the six-bit address part was not treated as an address, but as a

constant to be used in the actual computation. Indirect addressing is considered below in a more general context.

In the CDC 160A, seven banks of 4,096 words each were added to the storage, thus complicating the specification of an address. The modes of addressing already available were retained, but several three-bit registers were added to contain the number of the bank (0-7) in which a designated address would be found, and different operations referred to different bank registers. Additional operations were provided so that the programmer could set the values of these registers as necessary. This later machine also provided for two-word instructions, in which the second word might be a 12-bit immediate operand as well.

Indirect Addressing. One way to address a memory larger than the address part of an instruction allows is to have the instruction address point to another address that stores the operand address. This facility, called "indirect addressing," was available on the CDC 160 and is available on most contemporary computers. Fig. 1 illustrates this situation on a hypothetical 16-bit computer with a 7-bit instruction field and a 9-bit address field that permits the direct addressing of only $512 (= 2^9)$ memory locations.

Indirect addressing can be used to address a memory of up to 65,536 words. In the example in Fig. 1, the program has placed the operand address 021326 (where we express addresses in octal for convenience) at a specific address (125) in the first 512 words of memory. If the instruction is, for example, an "add indirect" instruction, the address 125 is interpreted as an indirect address or pointer to the actual operand at location 021326. The address stored at 125 (namely, 021326) becomes the effective address.

Some systems allow multilevel indirect addressing. Thus, the number stored at 021326 may have a bit set that indicates that it, itself, is an indirect address that points to another location which contains the operand address. Indirect addressing may be combined in various ways with the use of index registers to produce complex addressing chains.

Index Registers. The concept of an index register, sometimes called a "tally register" or "base register" (see below), grew out of the B-line or B-register introduced on some of the earliest computers developed in England at the University of Manchester. This represented a major advance in computer design. Index registers are hardware reg-

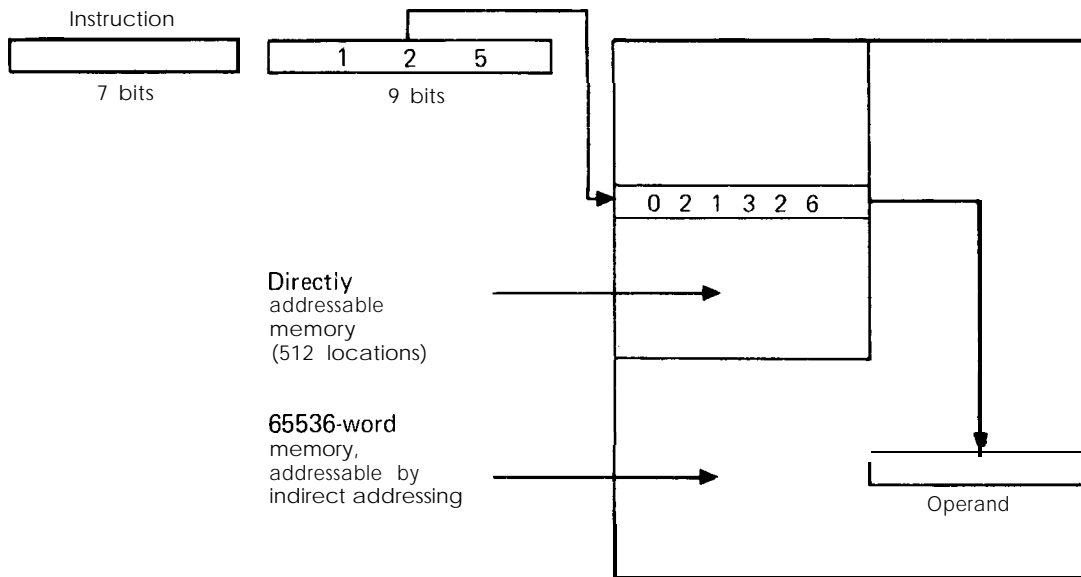


Fig. 1. Extension of addressing through indirect addressing.

isters that can be set, incremented, tested, etc., by machine instructions. Each instruction contains an indication as to whether its address is to be added to (or subtracted from) the contents of a designated index register to form the effective address. One of the main purposes, as suggested by the name, was to allow the effective address to be used as an index into a set of contiguous storage registers, commonly referred to as a "vector." Without changing the part of the address which was in the instruction itself, one could refer to one after another of the contiguous registers, merely by changing the contents of the index register successively. This replaced the more time- and space-consuming sequence of instructions which would normally put an instruction containing an address into an arithmetic register, modify it by ordinary addition, and then store it back to replace its former value. (This modified instruction was then executed, and it would refer to a different storage location.)

The use of index registers eliminated the need for modification of the instruction itself by allowing the index register to be modified by special instructions added to the computer for that purpose. With the advent of newer systems in which more than one task may be executing the same instructions at the same time, it has become very important that instructions not be modified during execution, since the modification by one task might be inappropriate for another task executing the same set of instructions.

General Registers. The use of variable-length instructions has also become much more widespread in recent years. The IBM System/360 or 370, for example, uses instructions that may take one, two, or three half-words for their representation. In the System/360 or 370, 16 *general registers* are provided, each capable of acting as an arithmetic register, a base register for relative addressing, or as an index register. (The fact that one cannot tell by looking at one of these registers whether its contents represents an ordinary number or an address has sometimes led to other problems, but this degree of flexibility is very useful.) An instruction might refer only to one or two of these registers, in which case only four bits would be needed in the instruction for each one, and it could fit in a half-word (16 bits).

A full-word instruction could accommodate one reference to a general register (4 bits) and a reference to a storage address. The latter could be a combination of a base register (4 bits), an index register designation (4 bits), and a 12-bit displacement, which could be used as a local offset from the contents of the base register. Fig. 2 illustrates the determination of an effective address from a System/360 or 370 instruction.

Relocation Registers. Many computers have one or more hardware relocation registers, which aid in the implementation and running of multiprogramming systems. An example is the CDC 6000 series. A number of different programs may be

ADDRESSING

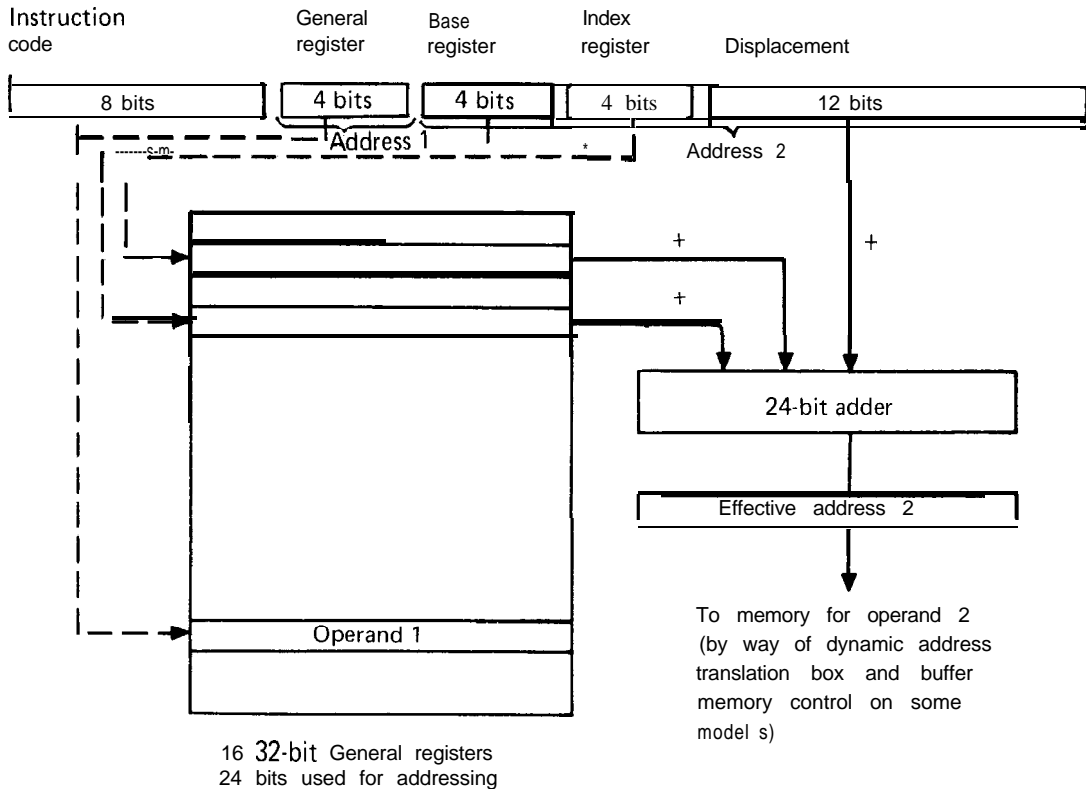


Fig. 2. Effective address calculation on IBM 360 and 370.

in the computer memory, each occupying a contiguous area. Thus, program A might occupy the area from 40,000 to 67,777, but this program (as well as all other programs in memory) is written and loaded into memory as if the area it occupies actually has the addresses 0 to 27,777. When program A is given control, the address 40,000 is stored in the hardware relocation register, and this constant is automatically added to all memory reference addresses while program A is running. The program could have been loaded anywhere else in memory, and can be loaded into different areas at different times. It will always produce the correct memory addresses, since all addressing is automatically made relative to the starting address of the area into which the program has been loaded.

In computers of this type, another hardware register will contain the "field length" or program size. Any attempt to reference beyond the area occupied by the program will be trapped, and an error condition will be signaled.

In a machine with two relocation registers, like the Univac 1108, a program may consist of two

segments: for example, a program segment and a data segment, which can be placed independently anywhere in memory. The starting addresses of each of the two segments are placed in the two relocation registers, and every effective address has an associated bit that specifies which relocation register is to be added.

A relocation register is quite different in nature from an index register or a register used as a base for relative addressing. The relocation register is a special hardware register whose contents can be accessed and changed only through the use of privileged instructions under control of the operating system.

Content-Addressable Storage. Content-addressable or associative memories are quite different in concept from the more conventionally addressed memories described above. In a content-addressable memory the data item itself contains a key, usually in a specified field. This key is, in effect, the address of the item. The key may be the whole data item itself. The desired data item is located by

means of an examination of all relevant keys. This could be done by software in a computer system with conventional memory addressing, but it would be quite slow.

In an associative or content-addressable memory, comparison circuits are used to provide a hardware-assisted and presumably very fast search through all data 'items to find the one that is addressed. Small memories of this type have been used to speed up address translation in virtual memory systems. Larger systems in which all addressing is associative have been proposed and some experimental models have been built.

The use of content-addressable memory was very expensive in terms of earlier technologies, but may prove practical with modern large-scale integration (LSI) technology. There are a number of important application areas in which associative memories would be very useful. The reader is referred to Hanlon (1966) for more information in this area.

B. A. GALLER AND S. ROSEN

SOFTWARE ASPECTS

Corresponding to each hardware aspect of addressing there must be one or more techniques by which the programmer specifies addresses in his program.

Absolute Addressing. In the earliest and most elementary programming systems a programmer would assign instructions and data to locations in memory, and instructions would refer to absolute locations in memory. Thus, using a decimal computer for convenience, a programmer might write

267 ADD 3256

and, as a result of the eventual loading process, the instruction ADD 3256 would appear in location 267. It was the responsibility of the programmer to make sure that the appropriate data word was in location 3256 at the time the program was to be run. These are absolute addresses in that 267 always represents the same physical location in memory and 3256 similarly represents a specific physical location.

Relative Addressing. Some of the first advances in programming involved permitting the programmer to write programs or parts of programs without having to be aware of the absolute physical

locations in which the instructions and data were to be stored. One of the early approaches to this goal was by way of regional or relative programming. A programmer, or several programmers, might decide that the program would be divided into a number of regions, A, B, C, D, etc. Addresses would then be relative to the start of a region. A programmer might write

A5 ADD B15

to specify that an instruction located in the fifth location in region A is to add (to the accumulator) the data located in the fifteenth location in region B. A translator and loader would eventually take all regional addresses and convert them into absolute addresses.

There are a number of important advantages to this procedure. The programmer does not have to make arbitrary decisions about how large the regions are going to be. Separate sections of the program can be written independently, and unexpected or undesirable interactions can be avoided.

Symbolic Addressing. It was a relatively short step from regional addressing to free symbolic addressing. In the typical assembly system the programmer may write

INCR ADD ALPHA

and leave it up to an assembler to decide where the instruction INCR is to be placed. Somewhere else in his program he would have a data item named ALPHA.

Indirect Addressing. In a computer that allows indirect addressing, the programmer typically indicates an indirect address by adding a character, such as *, to the absolute or symbolic address. Thus,

INCR ADD ALPHA*

would indicate that the effective address is not ALPHA but is in the location specified by ALPHA

Indexing. If an index register is to be used in calculating an effective address, this is normally specified following the instruction address. For example,

ADD A,4

indicates that the contents of index register 4 is to be

ADDRESSING

added (subtracted on some computers) to A to determine the effective address. Indexing can be combined with indirect addressing so that

ADD A*,4

would specify the effective address as the sum of the contents of location A and index register 4.

Higher-Level Languages. The development of higher-level programming languages has relieved the programmer of the responsibility for many aspects of memory management. However, that responsibility must reside somewhere: Either the programmer or the language processor and operating system must take on the responsibility for allocating space for instructions and data and for producing the programs that make appropriate use of the addressing structure of the computer. The software features of addressing discussed in this article are therefore mainly of interest to the assembly language programmer. The programmer who writes in a higher-level language such as Fortran or Cobol usually does not have to be aware of the details of memory addressing in the computer on which his program will run, but he may be sure that the Fortran or Cobol compiler is very much aware of these details, and usually expends a great deal of effort to take advantage of the memory-addressing hardware features provided on the computer.

S. ROSEN

VIRTUAL MEMORY

Overlays. Many programs are too long to fit into the space in main memory that can be allocated to them at run time. In a uniprogramming system this will be true when the amount of space required by the program is greater than the total memory available to problem programs. In a multiprogramming system it may be true because the amount of space that is needed is more than the operating system is willing to allocate to this particular program. In either case, it becomes necessary to break the program up into sections, segments, or overlays so that the entire program need not be in main memory at the same time. The term "folding" has sometimes been used for this process.

In many systems the programmer has the responsibility for breaking his program into overlays and for providing the loading instructions that bring necessary overlays into main memory as they are needed. Many software systems provide aids to overlay planning. The user can name his overlays so

that all symbolic addresses in an overlay will be automatically tagged with a special identifier that indicates which overlay they belong to.

A loader or linkage editor creates an object program organized as a set of overlays and a root segment containing information about the overlay structure. The root segment is loaded into main memory along with the segments needed to get the program started. Any reference to a symbolic address in a segment that is not in main memory causes a call on the supervisor to load the required segment, overlaying other segments if necessary.

There have been a number of efforts to produce software systems that provide automatic folding of programs. In such systems a programmer would write a program as if there were enough main memory to contain the whole program, and the software system would organize the program into overlays to fit the actual amount of storage that would be available. Efforts to produce software systems of this type date back to the earliest computers, but none has been particularly successful.

Most workers in the field feel that hardware assistance of some kind is necessary. Such hardware assistance is provided in the so-called virtual memory systems that first made their appearance around 1959 and which are becoming increasingly popular in the 1970s.

The Atlas System. The Atlas computer was probably the first virtual memory system. Its designers called it a single-level storage system. The idea was that a programmer would program as if all available memory were on a single level and directly addressable, whereas in fact memory was on two levels. In the Atlas the two levels were drum and core.

A program for the Atlas could be written as if it were to run in a homogeneous memory consisting of $2^{20} = 1,048,576$ words. Memory was organized into pages of $2^9 = 512$ words each. The physical core memory might consist only of 32 or 64 such pages. However, the "address space" (i.e., the addresses that a user could address) consisted of $2^{11} = 2,048$ such pages. Thus, an address in the Atlas consisted of an 11-bit page number and a 9-bit number indicating the location within the page.

A hardware page-address register is associated with each physical page (or "page frame," as it is sometimes called). A typical running program might consist of 50 pages, of which 20 pages at a particular time would be located in core memory and the other 30 located on the drum.

Each page of the program represents a set of 5 12 consecutive addresses with the same page number (i.e., the same 11 leftmost bits). The program page number is kept in the page address register of the physical page that is occupied by that program page. Thus, any program (or logical) page may occupy any physical page, and it may occupy different physical pages at different times during the running of the program.

Assume now (see Fig. 3) that the next instruction to be executed refers to an operand whose address (in octal) is 023 1443. This is a reference to location 443 in page 23 1. Note that core memory of the machine contains nowhere near 231 pages, and there are only 50 pages in the program being executed. The programmer does not have to confine his program to the first 50 pages or to any contiguous block of 50 pages. He can use any areas in virtual memory that are convenient. Thus, the programmer does not have to know beforehand how long his code areas and his data areas are going to be. He can break his program up into segments and place the segments far enough apart in virtual

memory so that he will be sure that their memory allocations will not overlap. There is no point at all to scattering a program at random over a large virtual memory; in fact, such programs will usually perform very inefficiently. One wants a very large virtual memory in order to be able to assign areas, whose ultimate size is not necessarily known in advance, to program and data modules that do not overlap and that form the structural units of a program. The segmented, two-dimensional virtual memories discussed below were introduced to make this type of modular programming more automatic and more convenient.

The page address registers form an associative or content-addressable memory. They are subject to a very rapid hardware scan to determine if one of them is page 231. If it is (say, if page 231 is in physical page 12, as in Fig. 3), then the operand sought is in physical location 12443, and the operand is fetched from that location. If, on the other hand, page 231 of this program is not in core memory, it must be fetched from the drum. An interrupt occurs, and the operating system initiates a transfer of that

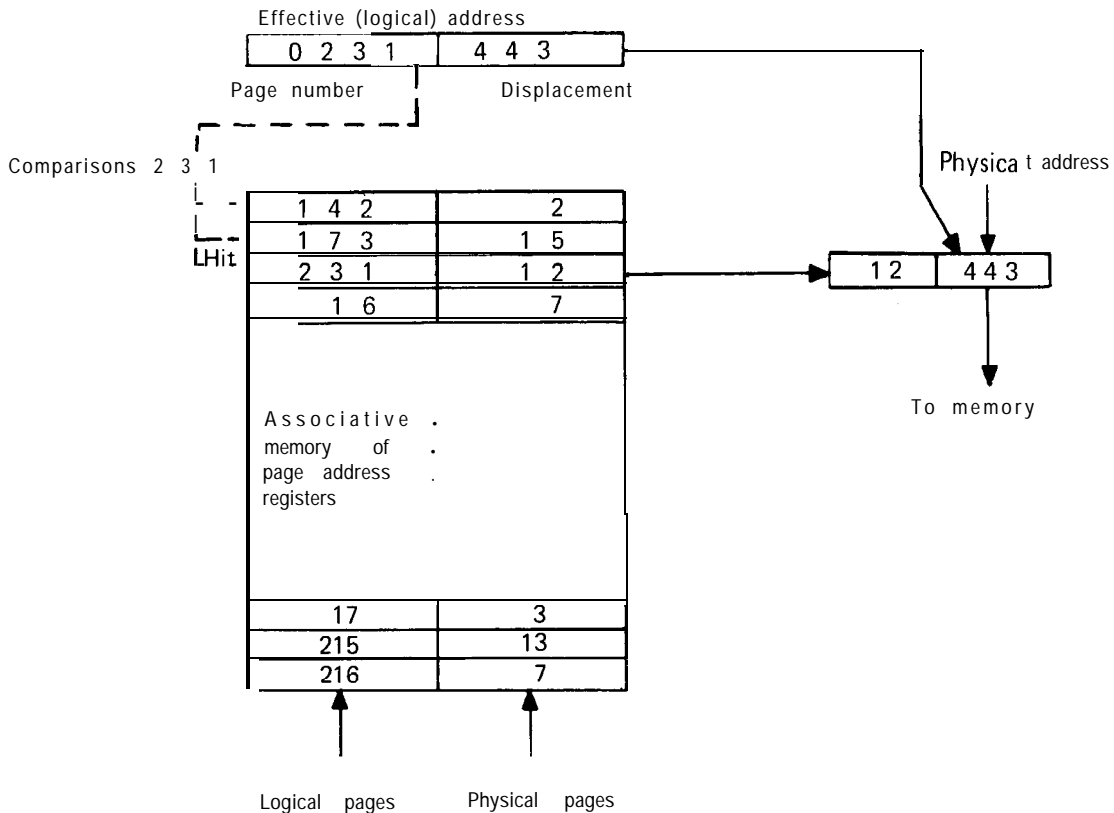


Fig. 3. Address translation on the Atlas computer.

ADDRESSING

page from drum into core. Assume that physical page 16 is available. The supervisor will cause program page 23 1 to be loaded into physical page 16, and will place the number 231 in the corresponding page address register. It then returns control to the program, which tries again to access an operand in virtual location 231443. This time it finds logical page 23 1 in physical page 16 and translates the address 23 1443 to 16443.

Segments and Pages. The Atlas system is an example of a one-dimensional or single segment virtual memory system. The programmer or the language processor must provide symbolic or absolute addresses in the one-dimensional virtual memory. Many of the classical storage allocation problems remain, although they are helped considerably by the fact that the virtual memory is much larger than the actual central memory of the computer on which the program is run.

From the point of view of program organization there are a number of advantages to a two-dimensional organization of virtual memory. Although two-dimensional virtual memory systems usually are multiprogrammed systems, it is convenient to think of each program in the multiprogrammed environment as if it were running in its own virtual memory.

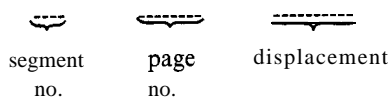
In such a system a program runs in a large virtual memory consisting of a number of segments. An address then consists of a segment name (or number) and a displacement relative to the beginning of the segment. This is somewhat analogous to the regional organization of programs in earlier computer generations and has some of the same advantages. The programmer or the language processor can assign programs and data to different segments without worrying about the relative position of the segments in the total addressing space. This is especially true if the segments are large enough so that possible segment overflow is not a problem. The segments themselves may be organized into pages.

A job (or process) is then represented in central memory by a segment table that provides a set of pointers to page tables corresponding to the active segments of the process. Each active segment will usually have one or more pages in memory. The actual address space or virtual memory is very large, and in most practical situations it consists mostly of unused space. Of the part of virtual memory that is actually used by a program, only a relatively few pages will be in central memory; the rest will reside on a paging drum (or disk) or in a backup mass storage system (usually disk storage). The segment

may serve as a unit of sharing among programs. The same segment (i.e., a pointer to the same page table) may appear in several segment tables that correspond to several jobs that are simultaneously active. The possibility of sharing segments was one of the strong motivations for the development of the segmented virtual memory systems.

The first and perhaps the only complete implementation of this type of virtual memory system was attempted in the Multics system developed at M.I.T. on the General Electric (now Honeywell) 645 computer. The actual addressing and address translation scheme used is too complicated to be discussed here. The reader is referred to **Organick** (1972).

IBM Virtual Memory Systems. The IBM 360/67 in 1965, and later the IBM 370 series in 1972, have popularized some of the concepts of virtual memory. In the standard 360/67 hardware systems the virtual memory associated with a process consists of 16 segments, and a segment consists of 256 pages, each of which has 4096 bytes. The relatively small size of the virtual memory loses some of the advantages associated with virtual memory systems, but it probably has the advantage of making memory addressing more manageable. In the IBM 360/67 system the 24-bit address field* contains a 4-bit segment number, an 8-bit page number, and a 12-bit byte address within the page. (The IBM system 370 provides as an option the use of a 5-bit segment number with a 7-bit page number. This will probably be the standard option used by the 370 virtual memory operating systems.)



The operating system maintains a 16- (or 32-) word segment table for each process that contains the pointers to the page table for each segment. The page table contains the physical address of each page that is present in main memory. These tables are automatically searched when a memory reference is made. Thus, if page 4 of segment 6 is in main memory starting at physical byte location 15000 (hexadecimal), a reference to address location

* An optional hardware modification provided for the use of a 32-bit address field that made it possible to use a 12-bit segment system and thus address 4096 segments. Several software systems, including TSS (Time Shared System) 360, took advantage of this extended addressing capability.

ADDRESSING

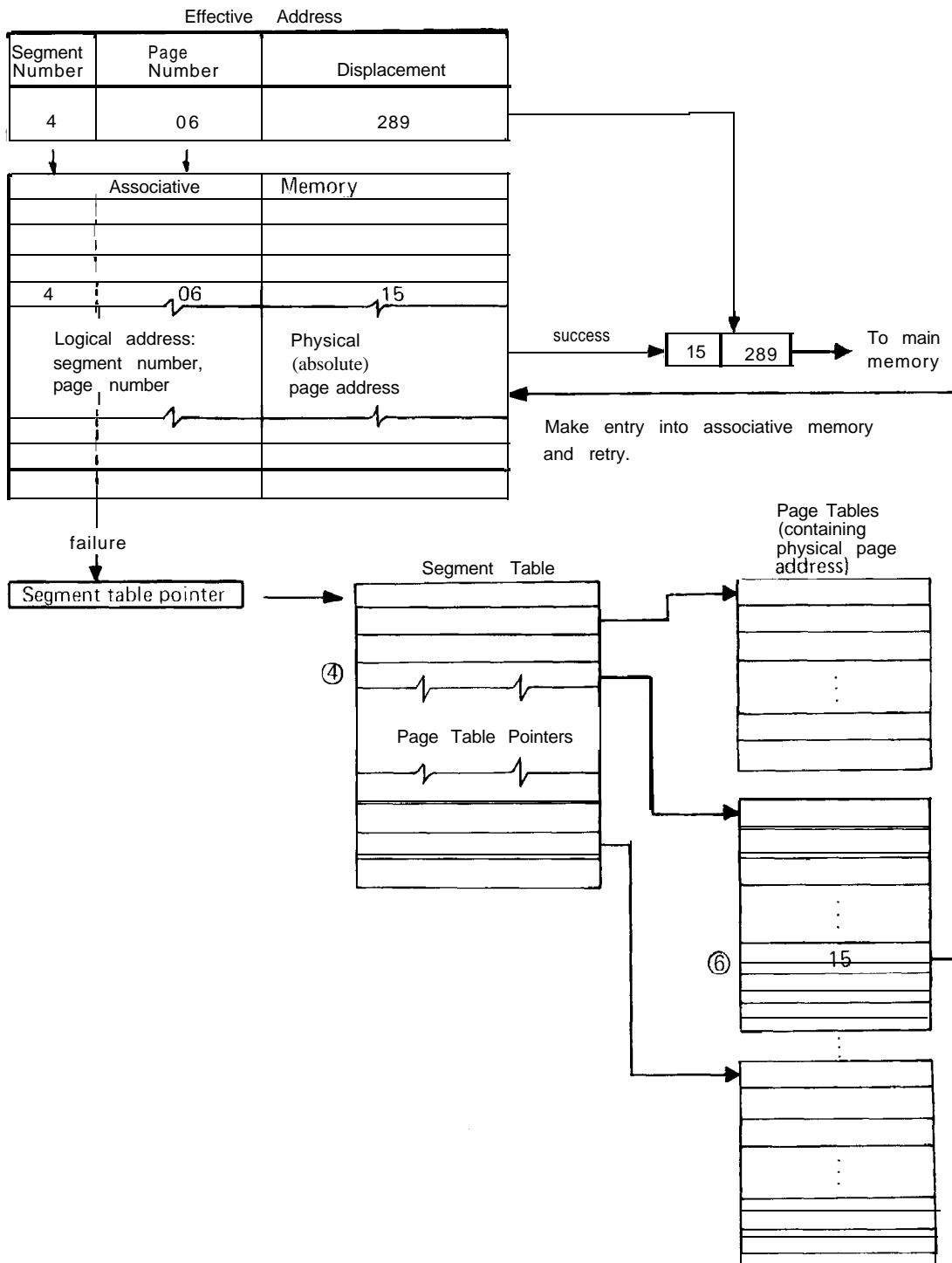


Fig. 4. Address translation on a virtual memory machine (IBM 360/67).

ADMINISTRATIVE-BUSINESS APPLICATIONS

406289 would cause the dynamic address translation hardware to search the segment table for segment 4 and retrieve the address of the page table for segment 4. It would search the page table for page 6 and retrieve absolute address 15. It would thus translate the segment/page address 406 into the physical address 15. (Actually 15000, but the three trailing zeros are understood.) The absolute address reference is thus 15289, and the operand is retrieved from that address.

Systems of this type usually have a small associative memory in which the most recent translations are stored (Fig. 4). Thus, the first reference to page 6 of segment 4 would proceed as discussed above, but the fact that 406 translates into 15 would be retained in the small associative memory. Then, so long as page 6 of segment 4 is one of the most recently referenced pages, its translation will be in the associative registers and the relatively slow address translation process does not have to be repeated.

Burroughs 5000 Series. In the Burroughs 5000 series and its successors, a job is represented in memory by a program reference table (PRT). Along with some data and other information, the PRT contains "descriptors," which are pointers to data segments and to program segments. A descriptor for a data segment (an array) contains the address of the beginning of the array and the length of the array. Any reference to the array is automatically checked, and an interrupt occurs if it is attempting to reference beyond the array bounds.

In these systems, one bit of each data word is reserved as a marker bit that marks the word as a datum or as a descriptor. This has been generalized on some systems by allocating several bits of each data word to provide information about the data type and format along with each individual word or item of data.

The Burroughs systems are virtual memory systems, based on the use of relatively small segments that are not broken into pages. A segment is moved as a unit between central memory and backing drum or disk storage. The operating system is a multiprogramming system. Each active job has some of its segments in core memory and the rest on the drum. Each segment descriptor in the program reference table has a presence bit, which indicates whether or not that segment is present in central memory. Any attempt to refer to a segment that is not in central memory causes an interrupt to the supervisor, requesting that that segment be loaded. The supervisor can load the segment into any

available contiguous area of memory that is large enough to hold it. If necessary, it can move out other segments. When the segment has been loaded, its new starting address in central memory is placed in its descriptor, and the program that referred to the segment can be restarted.

S. ROSEN

REFERENCES

- 1966. Hanlon, A. G. "Content-Addressible and Associative Memory Systems-A Survey," IEEE Transactions in Electronic Computers, August.
- 1971. Bell, C. G., and A. Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill.
- 1972. Organick, Elliott I. *The Multics Systems-An Examination of Its Structure*. Cambridge: M.I.T. Press.

ADMINISTRATIVE-BUSINESS APPLICATIONS

For *articles on related subjects see **APPLICATIONS, COMPUTER; COMPUTER NETWORKS; COMPUTER SYSTEMS; CREDIT SYSTEM APPLICATIONS; DATA COMMUNICATION NETWORKS; DATA PROCESSING; POINT-OF-SALE TERMINAL; PROCESSING MODES; SCIENTIFIC APPLICATIONS; SIMULATION; and TIME SHARING.**

Administrative or business applications refer broadly to those computer-based systems used in the execution and management of the basic operations of an enterprise. While these systems are most frequently found in commercial and industrial organizations, they have direct counterparts in government and not-for-profit organizations. Administrative or business applications are distinguished from process control, engineering, information retrieval, and management information systems by the category of operation they are intended to serve.

Mechanization of these applications has been, and continues to be, the most frequent reason that business organizations install computers. In the early days of computer use, routine applications such as payroll and inventory normally were the first to be computerized. Today, as organizations seek to solve more difficult business problems and to gain control over broader and more closely integrated areas of

their operations, highly complex computer applications are being undertaken. For example, distribution management systems are being developed to integrate finished goods inventory control, warehousing, and transportation, and new order-entry systems do the same for requirements planning, production scheduling, and sales analysis. Whereas the high cost of early computers limited their installation to large organizations, the extension of the minicomputer's capability-beyond its previously scientific or process control orientation-is making it possible for thousands of small organizations to computerize their operations.

This article describes the evolution of administrative and business applications over the past 20 years and discusses specific applications that have been and are being computerized in each of the four major areas of business and administrative operations. Finally, the References at the end of the article list sources of additional information about such systems.

Evolution of Administrative and Business Applications. In order to establish a frame of reference for reviewing specific systems, this section outlines briefly how such systems have evolved, and why, over the past 20 years.

EARLY SYSTEMS. When the first computers were installed in commercial and industrial organizations in the 1950s, they were most often used to automate office functions such as payroll, invoicing, and inventory accounting. This was a logical step in the evolution of these activities, for several reasons:

1. These activities were well understood and reasonably well documented as a result of existing accounting and auditing requirements, both of which made computerization easy.
2. The precomputer step-by-step method of carrying out these activities suited the capabilities of early computers, which could only process information serially from card files or magnetic tape. Moreover, in many cases these activities were already being processed on punched card equipment, making them easy to convert to a computer.
3. Computerization of these functions was readily justified by clerical staff reduction and by savings brought about by replacing more expensive punched card equipment.

These early computer systems processed data in batches; i.e., they executed one program at a time and handled transactions one by one after they had been sorted into numerical sequence. In general,

they were simply more efficient electronic outgrowths of the punched card systems that were initially installed in the early 1930s. In spite of their greater speed and capability, they still retained many limitations of their predecessors: They required substantial manual intervention and their impact was confined to narrow functional areas of business. The applications they processed were similarly limited in scope.

ADVANCES IN SYSTEMS. Many of the administrative and business applications that are commonly processed on computers today differ greatly from the earlier accounting-oriented applications. Notably, they are **broader in scope**: Whereas early applications dealt with a single activity, current systems frequently consist of several integrated subsystems covering a broad segment of the business.

They are more **responsive**: Early systems performed simple calculations, printed out 100% of the transactions they processed, and were run usually at fixed intervals (weekly or monthly). Today's systems perform complex calculations, screen results so that only those that exceed a range of acceptable performance are printed out or displayed, and do so daily or even hourly, with the result that important information is made available in time for corrective action to be taken.

They **serve many more users**: Initially, an employee who wanted to run a computer program did so at a computer center, or he sent his job to the center to be processed. Now, through easy-to-use terminals, computers can be accessed by large numbers of users who may be close by or thousands of miles away.

They **offer a variety of new benefits**: Traditionally, reduction of clerical costs was the primary benefit anticipated from automation of administrative or business functions. Today, computerized applications have the potential to provide information that can result in a wide range of benefits, from significantly improved customer service to tighter managerial control over major segments of the organization.

Some of the commercial systems in use today illustrate these advances. For example, by means of a Touch-Tone telephone or a small remote terminal, an airline reservation clerk can call upon a central computer to check a customer's credit rating; the computer can notify him in a matter of seconds whether the credit card is stolen or the account is delinquent. Similarly, the sales clerk in a local store can be connected to a computer by telephone and can be told by computer-produced voice response

ADMINISTRATIVE-BUSINESS APPLICATIONS

EXAMPLE: CONVERSATIONAL ORDER ENTRY

QUESTION: COMPUTER GENERATED DISPLAY	RESPONSE BY TERMINAL OPERATOR
ORDER ENTRY OE ORDER STATUS INQUIRY OS PAYMENT POSTING PP (etc.)	
CUSTOMER NAME OR NUMBER	OE
SMITH DATA PROCESSING CO ATTEN P.J. ANTHONY 635 JACKSON ST LOS ANGELES CA 96402	SF51 90 (or) SMITH DATA PROCESSING
SHIP VIA CF IS THIS CORRECT?	C (correct)
ENTER MACHINE NUMBER, QUANTITY	129, 1
SPECIFY (1 AND 2 REQUIRED, 3 AND 4 OPTIONAL)	
1. VOLTAGE (AC, 1 -PHASE, 60 CYCLE) A LOCKING PLUG, 115V #9880 B LOCKING PLUG, 208V #9884 C NON-LOCKING PLUG, 115V #9881 D NON-LOCKING PLUG, 208V #9885	
2. CHARACTER ARRANGEMENT A1 1	
3. ACOUSTIC COVER FOR CARD TRANSPORT (FIELD INSTALLABLE) #9014	
4. CARD I/O ATTACHMENT #9619	
9882 INVALID FEATURE NUMBER, REENTER CORRECT VOLTAGE AND PLUG	9882, A1 1, 9014
FEATURE 9014 FACTORY OR FIELD INSTALLED?	9880
SPECIAL FEATURES	FACTORY
NOTE: MAXIMUM COMBINATIONS ARE INDICATED BY X IN VERTICAL COLUMNS	
ACCUMULATOR #1020 x x x x ACCUMULATE PROGRAM LEVELS #1025 XXXX AUXILIARY STORAGE #1201 x CARD I/O ATTACHMENT #7503 x INTERPRET #4601 x FEED, VARIABLE LENGTH #3950 PRODUCTION STATISTICS #4802 XXX: SELF CHECK NUMBER MOD 10 #7061 x x x	
RPQ 'S ? (special engineering)	1020, 1025, 4802, 7061
DELIVERY REQUESTED	NO
ADDITIONAL REMARKS?	MAY 29, 1974
MORE ADDITIONAL REMARKS?	CALL MR. ANTHONY BEFORE DELIVERY TO RESERVE FREIGHT ELEVATOR
END OF ORDER ENTRY PROCEDURE?	NO
ORDER ENTRY OE ORDER STATUS INQUIRY OS PAYMENT POSTING PP (etc.)	YES

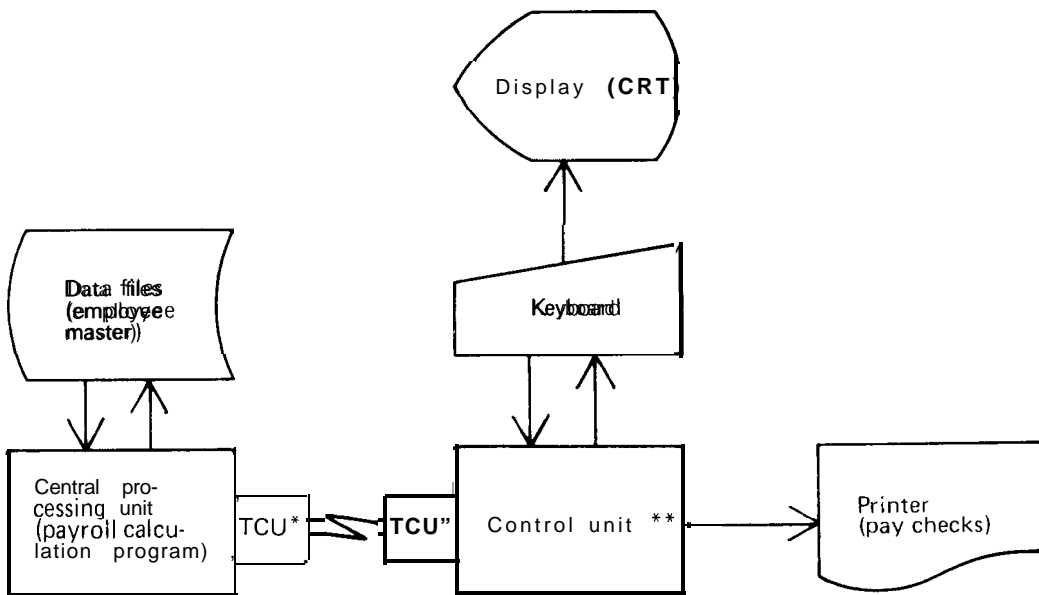
ADMINISTRATIVE-BUSINESS APPLICATIONS

whether a product requested by a customer is available, where it is located, and when it will be shipped. In a third example, manufacturers can install computers for conversational order entry. Instead of writing or typing out all detailed specifications about a machine to be ordered, the terminal operator tells the computer he wants to enter an order for a specific model of machine. The computer then asks a series of questions about the order, which the operator answers in turn. When the dialog is completed, the full specifications are recorded in the computer ready for matching with production and shipping schedules. Exhibit 1 illustrates the type of dialog that takes place between terminal operator and computer in conversational order entry.

FOUR STAGES OF EVOLUTION. The evolution of administrative and business systems to their current status has followed the expanding hardware and software capabilities of the computer. Exhibit 3 describes four broad stages of computer development and the application characteristics that correspond to those stages. Briefly:

1. *Basic batch* is the least complex level of computer processing. At this level, application systems are normally made up of small programs that are run through the computer one by one and which can process transactions only from sequential files.

2. *Expanded batch* programs perform complex computations and produce reports that analyze performance, not just report it as in basic batch systems.



* — Transmission control unit.

** — Contains arithmetic, logic, and primary storage units. Auxiliary storage (e.g. disk unit) would be attached to and controlled by this unit also.

CENTRAL (COMPUTER) SITE

1. Receive data from remote location
2. Calculate payroll using programs and data files (e.g. employee master) located at central site
3. Update payroll files
4. Transmit payroll data to remote location

REMOTE (TERMINAL) LOCATION

1. Input payroll data (e.g. hours worked by job class) needed for central computer payroll calculation
2. Control Unit edit of data, operator correction of invalid or incomplete data
3. Transmit to central site computer
4. Receive payroll data from central site computer, format and print pay checks

Exhibit 2

ADMINISTRATIVE-BUSINESS APPLICATIONS

Larger programs, further automation of manual functions, and a small capability for processing transactions that occur in random sequence are characteristics that distinguish expanded batch from basic batch systems.

3. *On-line inquiry* application systems result from adding to expanded batch systems the capability to immediately access, by terminal, any record that is stored in the disk files attached to the computer. Processing of transactions that are not in numerical sequence is also possible at this higher level of systems complexity.

4. *Distributed computing* systems consist of combinations of large central computers, data communications networks, and remote terminals that enable terminal operators located remotely from the central computer to carry out complete operations. For example, an employee's paycheck can be prepared while the operator is sitting at the terminal. Earlier, less sophisticated terminal systems would have required the operator to send the data (hours worked) to the central computer, where it would have been processed (in a batch mode) and the paycheck data later (hours or days) retransmitted to the remote terminal.

Distributed computer systems have the following characteristics :

(a) Processing is in line, completed while the operator is at the terminal.

(b) They are hierarchical; at some point in the processing of a program, or under certain conditions, data are transmitted from the terminal to a central computer for processing that is not possible at the terminal.

(c) The remote location may, but need not, have either auxiliary storage or the capability to execute programs within the terminal.

Exhibit 2 illustrates a distributed computing system used in payroll processing. The mode of processing is in line, with the terminal operator keying in the data required by the central site computer program, correcting errors caught by the terminal edit program, and transmitting the data to the central site computer. The central computer calculates the payroll and retransmits paycheck data to the remote terminal, where it is printed.

Exhibit 3 illustrates that as the computer's capabilities have progressed from basic batch processing to on-line inquiry and distributed computing, commercial applications have also advanced from relatively simple, low-cost, narrow impact groups of

programs to highly complex, expensive, and far-reaching systems.

The remainder of this article will discuss the four major types of administrative and business applications : customer service, manufacturing, accounting and finance, and payroll and personnel. To aid in reading the exhibit that accompanies each of these four sections, customer service will be reviewed in detail, while the remaining three categories will be discussed less extensively.

Customer Service Systems.

Customer Service Systems. Customer service systems relate to the interaction between an organization and its customers. Processing orders, keeping track of inventory available to meet customer demand, collecting accounts receivable, and analyzing sales transactions are representative of the functions that fall in the category of customer service.

Customer service is an area of vital concern in most industrial and commercial organizations both because of its great profit potential and because of the difficult management problems it poses (for example, how to provide a high level of service to customers without reducing profit margins). Increasingly, computerized customer service applications are helping managements resolve these problems and achieve better control over this area of operations. For example, customer service applications today can provide the information needed to answer important questions such as the following:

Is the amount of inventory on hand too much or too little?

Which products in the line are contributing to profitability?

Which customers are profit generators?

Are salesmen merely selling as much business as they can, or are they selling profitable business?

Is the level of accounts receivable too high?

Are bad debt write-offs above industry norms? Are they increasing?

Although some of the earliest business computer applications were related to customer service, only very recently have computer capabilities advanced to the point where large-scale improvements in this area of business operations are possible. Today, many of the most sophisticated and expensive computer systems being installed in commercial and industrial organizations are in customer service. In addition, industry sources estimate that one-third of the computers installed throughout the world pro-

EVOLUTION OF ADMINISTRATIVE AND BUSINESS SYSTEMS

Application Characteristics	Least Complex —————→ Most Complex			
Category of hardware system	Basic Batch • electronic unit record plus magnetic tape	Expanded Batch • electronic unit record, magnetic tape, small capacity disk storage	On-Line Inquiry • electronic unit record, larger disks, typewriter, and CRT terminals	Distributed Computing • large disks, remote terminals (mini-computers, CRTs), data communications devices
Cost to develop an application system	Less than \$50,000	————— increasing to —————	————— increasing to —————	\$500,000's to millions
Time required for development	Several man months	————— increasing to —————	————— increasing to —————	Several man years (2-5 plus)
Degree of integration with other application systems	Low • normally 1 department served	More than 1 department served	Several departments served	High • many departments served
Degree of difficulty of development and implementation	Simple • mostly internal to data processing department	Relatively simple • some technical problems	Relatively difficult • operational problems in implementation	Difficult • often large operational as well as technical problems
Criticality of computerized system to departmental operation	Low • manual backup always possible	Medium manual backup usually possible	Medium • manual backup difficult	High manual backup almost impossible
Organization level at which commitment needed for development and implementation success	Manager of department requesting the system	Manager of department requesting the system	Vice president	President, top managing executive
Type of benefit	Clerical savings	Primarily clerical savings, some improvement in management control	Improved customer service, improved management control	Cost reductions, improved service, improved management control
Magnitude of benefit	Small, but demonstrable and immediate	————— increasing to —————	————— increasing to —————	Large, difficult to quantify in advance, gained over time
Amount of user participation required in systems development	Limited, generally not critical	Needed in design phases	Needed in all phases	Extensive, critical to project success
Amount of user participation required in systems operation	None	Limited to coding of input data	Moderate • for inquiry operation	Extensive, user runs the system from terminals
Response time of system to user request for information	Week(s)-days	Days	Minutes • seconds	Minutes • seconds
Historical time frame of first systems	Early 1950s	Mid-1 950s	Late 1950s	Late 1960s

Exhibit 3

ADMINISTRATIVE-BUSINESS APPLICATIONS

cess customer service applications-another indication of their importance.

Exhibit 4 lists the applications that make up the customer service group and illustrates how sophistication and complexity increase as computer capabilities progress from basic batch to distributed computing.

BASIC BATCH. In most organizations, automation of customer service applications began at a relatively low level of complexity; i.e., in the form of basic batch systems. Despite their lack of sophistication, such systems were often able to produce important benefits.

The second column of Exhibit 4 describes the types of customer service applications typically executed in a basic batch mode. In this mode, the computer performs a number of calculating and record-keeping functions formerly performed manually. In order processing, for example, after the information about an order is entered into the computer (via key-punched cards or magnetic tapes or disks), the computer prepares shipping notices, warehousing picking lists, and later invoices showing items shipped and back ordered, price and discount amounts, and shipping, taxes, and other charges. After an order is processed, the same records that were used to prepare invoices can be sorted and rerun to generate weekly stock status reports showing beginning balance, amounts received, shipped, back ordered and on order, and balance on hand. The same records can be sorted again and processed to produce sales analysis reports listing sales volume by product, customer, and salesman; and finally, the records can be used to prepare aged trial balances and accounts receivable statements. Exhibit 5 illustrates examples of an aged statement and trial balance.

Despite their low level of complexity, basic batch customer service systems can be very successful. In many instances the introduction of such systems has permitted sizable reductions in clerical staffs. In almost all instances, organizations have benefited from faster processing and more accurate, up-to-date, and complete records about inventory levels, sales transactions, and accounts receivable status. With this information, managements have been better able to answer many of their basic questions about product movement, customer purchases, and sales force effectiveness.

EXPANDED BATCH. The desires to perform more sophisticated analyses and to further reduce manual intervention in processing customer service functions were the forces that led many organizations to reprogram their basic batch systems into systems of

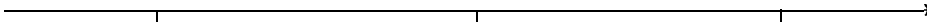
greater complexity and capability. This second stage of customer service systems development is described in Exhibit 4 under the column headed Expanded Batch.

Many operations that were manual in the basic batch environment, such as credit checking, release of back orders, and handling of special billing instructions are automated in expanded batch customer service systems. Order entry, which was normally carried out by key-punching cards in the basic batch mode, is automated in numerous systems by the use of **OMR** (optical mark reading) and **OCR** (optical character recognition) order documents, which are filled out by the salesmen at the time the orders are taken and are then fed directly into the computer for processing. Rate and routing tables are frequently computerized, making calculation of freight charges and preparation of transportation documents possible. Computer printing of collection follow-up letters further reduces manual intervention.

In the basic batch mode, computerized sales analysis consisted merely of breakdowns of sales volume by customer, product, and salesman. In expanded batch systems, as historical sales records and product cost information are made part of computer files, comparative analyses can be made of current versus historical sales volume and profitability by customer, product, and salesman. Exhibit 6 shows examples of such reports. In addition, inventory systems are expanded to include programs for statistical sales forecasting and for the calculation and reporting of economic order quantities (**EOQ**) to aid inventory managers. Inventory levels are determined on the basis of current and historical sales records, price, and cost data, enabling many organizations to reduce investment in inventory without sacrificing customer service.

ON-LINE INQUIRY. By the mid- to late 1960s, many organizations had developed comprehensive expanded batch customer service systems. The data files of these systems contained detailed information about many aspects of the customer service function; however, the information was not readily available because it was usually stored on a reel of magnetic tape located in a library away from the computer. Obtaining information from these data files to answer unscheduled requests (e.g., What is the balance on hand of a certain part in inventory?) took hours to accomplish and was a costly, disruptive operation.

On-line inquiry systems resolve the problem of inaccessibility by keeping primary data files (about customers, products, and orders) continuously

<u>CUSTOMER SERVICE SYSTEMS</u>				
	Least Complex  Most Complex			
Application	Basic Batch	Expanded Batch	On-Line Inquiry	Distributed Computing
Order Processing	<ul style="list-style-type: none"> Preparation of billing documents Warehouse picking lists Customer invoices 	<ul style="list-style-type: none"> Automated order entry OCR (optical character recognition) OMR (optical mark recognition) Order editing Computation of freight charges Handling of invoicing exceptions 	<ul style="list-style-type: none"> Credit checking Stock availability checking Order status checking 	<ul style="list-style-type: none"> Conversational order entry Multiple location stock availability checking Automatic order transmission to shipping warehouse On-line invoicing
Inventory • Finished Goods	<ul style="list-style-type: none"> Weekly stock status reporting 	<ul style="list-style-type: none"> Daily exception reporting Sales forecasting Simple EOQ calculations 	<ul style="list-style-type: none"> Stock status inquiry handling 	<ul style="list-style-type: none"> Continuous updating of inventory records Multiple location balancing of stock Complex EOQ calculations
Accounts Receivable	<ul style="list-style-type: none"> Preparation of aged trial balances and monthly statements 	<ul style="list-style-type: none"> Preparation of follow-up letters 	<ul style="list-style-type: none"> Account status inquiry handling 	<ul style="list-style-type: none"> On-line cash posting and account maintenance Automated scheduling of follow-up activities
Sales Analysis	<ul style="list-style-type: none"> Breakdown of sales volume by product, customer, and salesman 	<ul style="list-style-type: none"> Analysis of sales profitability by product, customer, and salesman 		

Note: System component features listed in one column will be found where applicable (though not listed again) in all columns to the right {e.g., order processing application feature "order editing" under "expanded batch" will be part of "on-line inquiry" and "distributed computing" systems}

Exhibit 4

AGED TRIAL BALANCE									
PAGE 1									
DATE 7/17/74									
CUST NO	CUSTOMER NAME	TELEPHONE NUMBER	LAST PAY CREDIT	TOTAL	CURRENT	OVERDUE	ACCOUNTS		
			MO DA YR LIMIT	OUTSTANDING	AMOUNT	30 DAYS	60 DAYS	90 & OVER	
108	ALLEN & CO	415-378 1089	2 16 74	\$15,000	\$ 7,296 35	\$ 6,919 77	\$ 376 58		
165	ANDERSON AUTO SUPPLY	408.2866741	1 28 74	2,500	1,665 49	1,665 49			
178	ANDREWS AND SONS INC	408-262-2074	2 05 74	750	146 64		\$ 146 64		
189	ARGONAUT ENGINEERING	415-867 2506	12 27 73	2,000	3,125 41	2,111 30	611 54	312 13	\$ 9 0 44
247	BERKLEY PAPER CO	408-251-4 189	2 21 74	6,300	5,289 00	1,855 50	2,652 45	1,400 06	51 00
252	BEST DISTRIBUTION CO	408-296-1667	10 06 73	1,000	765 44	3 25			762 197
FINAL TOTALS				\$36,241 33	\$21,085 31	\$5,601 57	\$3,831 82	\$998 63	

The combination of an old receivable and no current payment indicates a problem requiring management attention

Note: This type of trial balance—analyzed and listed by age of receivables—is very difficult to prepare by hand or bookkeeping machine.

In this accounts receivable system, payments are posted directly against a specific invoice, not just against an open balance

STATEMENT					STATEMENT						
CUSTOMER NO 554386					DATE 8/30/73						
HITTON CORPORATION 138 MARSHALL DR PO BOX 851 LONG PORT CALIF 94134					HITTON CORPORATION 138 MARSHALL DR PO BOX 851 LONG PORT CALIF 94134						
DATE		INVOICE	REFERENCE	DESCRIPTION	AMOUNT	DATE		INVOICE	REFERENCE	DESCRIPTION	AMOUNT
MO	DAY	YR	NUMBER			MO	DAY	YR	NUMBER		
09	08	73	185163		PRIOR BALANCE	\$2,565 46	09	08	73	185163	\$2,565 46
09	10	73	075126		INVOICE	1,685 91	09	10	73	075126	1,685 91
09	16	73		091531	PAYMENT	1,856 00CR	09	16	73	091531	1,856 00CR
09	30	73			LC ADJUSTMENT	13 00CR	09	30	73		13 00CR
					LATE CHARGES	6 96					6 96
CURRENT AMOUNT					30 DAYS	CURRENT AMOUNT					30 DAYS
\$1,693 91					\$696 46	\$1,693 91					\$696 46
					60 DAYS & OVER						60 DAYS & OVER
					BALANCE DUE						BALANCE DUE
					\$2,390 37						\$2,390 37
PLEASE RETURN THIS PART WITH YOUR PAYMENT						PLEASE RETURN THIS PART WITH YOUR PAYMENT					

Computed automatically, based on overdue balance

Exhibit 5. Aged trial balance and statement.

COMPARATIVE ANALYSIS OF SALES BY ITEM									
PERIOD ENDING 10/31/74									
PAGE 12									
ITEM NO	DESCRIPTION	CURR THIS YR	PERIOD LAST YR	QUAN. YR	PCT CHG	YTD THIS YR	QUANTITY LAST YR	PCT CHG	
824634	020068 OVERHAUL GASKET	10	14	29-	90	98	8-		
624832	17D0011 BELT DYNAMIC FAN	190	150	27	1,820	1,905	4-		
624901	DMK6448 HUB ASSEMBLY J2	1-	a	120-	18	18	0		

Indicates product movement trend

SALES ANALYSIS BY ITEM FOR EACH CUSTOMER									
PERIOD ENDING 03/31/74									
PAGE 29									
CUST NO	ITEM NO	CUSTOMER/ITEM NAME	SOLD THIS PER 100 QUANTITY	AMOUNT	PROFIT PRCNT	SOLD THIS YEAR QUANTITY	AMOUNT	PROFIT PRCNT	
0667		CONTINENTAL ELECTRIC CO							
	411116	B500 TWINLITE SOCKET 8	320	192.45	14	1,200	720.03	14	
	411122	B506 SOCKET ADAPTER BROWN	1,000	320.05	6	5,200	1,677.34	9	
	411173	C151 SILENT SWITCH IVORY	25	30.00	12	865	1,038.23	12	
		CUSTOMER TOTALS	542.50	12		3,435.60	12		

Shows volume and overall profitability by customer

COMPARATIVE SALES ANALYSIS BY CUSTOMER									
FOR EACH SALESMAN									
PERIOD ENDING 07/31/74									
PAGE 43									
SLMN NO	CUST NO	SALESMAN /CUSTOMER NAME	THIS PERIOD THIS YEAR	THIS PERIOD LAST YEAR	YEAR-TO-DATE THIS YEAR	YEAR-TO-DATE LAST YEAR	PRCNT CHNG		
10		A R WESTON							
	1426	HYDRO CYCLES INC	3,210.26	4,312.06	10,010.28	9,000.92	11		
	2632	RUPP AQUA CYCLES	7,800.02	2,301.98	20,822.60	11,020.16	99		
	3217	SEA PORT WEST CO	90.00CR	421.06	593.10	900.00	34-		
		SALESMAN TOTALS	10,920.28	7,035.10	31,425.98	20,921.08	50		
12		HT BRAVEMAN							
	0301	BOLLINGER ASSOCIATES	100.96	0.00	100.96	70.00	44		
	1941	MACK HARDWARE CO	4,201.85	860.82	13,922.68	11,866.22	17		
	2601	RECREATIONAL UNLIMITED	180.60	1,420.96	8,029.22	1,200.00	28-		
		SALESMAN TOTALS	4,483.41	2,281.78	22,052.86	23,136.22	5-		
		FINAL TOTALS	98,473.26	91,805.82	656,002.17	598,771.49	10		

This salesman is generating fewer sales than last year and is below the average of all salesmen

Exhibit 6. Sales analysis reports.

ADMINISTRATIVE-BUSINESS APPLICATIONS

mounted on disk drives attached to the computer. As a result, the data needed to answer questions are always available to operators of typewriter or CRT (cathode ray tube) terminals connected to the computer. The introduction of such systems has made possible a number of significant improvements in customer service. For example, in many cases customer inquiries can now be handled while the customer is on the telephone, not hours or days later.

The fourth column of Exhibit 4 lists the customer service applications in which on-line inquiry is most often used. These include systems to check the status of various items, such as credit in a given account or stock levels in a warehouse. All other customer service programs, such as preparation of invoices and accounts receivable statements, continue to be processed in the expanded batch mode.

DISTRIBUTED COMPUTING. Today, computer systems are being installed in industrial and commercial organizations that extend the power of the computer to the user wherever he is located. Such systems-known as distributed computing systems-are complex and expensive. They can, however, offer benefits that outweigh their costs.

Being able to distribute the power of a large, central computer among many remote users is a relatively new capability in industrial and commercial applications processing, although it has existed for some time in the form of time sharing for mathematical and statistical problem solving. In customer service, in particular, such systems have permitted major improvements in management, control, and responsiveness.

The fifth column in Exhibit 4 outlines the customer service applications that can be processed in a distributed computing environment. Conversational order entry (the top item in the column) implies that the clerk enters an order by "talking" with the central computer via a terminal. The computer asks a number of questions about the order, and when all are answered the order is complete and ready for automatic transmission to the location from which the items will be manufactured or shipped. In this mode of processing, the central computer automatically checks all stock locations (not just the one at which an order is normally filled for a specific customer) before declaring an item out of stock. And when an order is shipped, the computer notifies the order clerk, who completes the order-processing cycle by preparing the invoice at his terminal.

Distributed computing systems are complex. They may take years to design and may cost hundreds of thousands (and sometimes millions) of

dollars to install. They require large central computers, complex software, and normally involve extensive communication and terminal networks. Nevertheless, they can be significantly less expensive than a number of decentralized stand-alone computers installed to do the same jobs.

In addition, distributed computing systems can help bring about major improvements in customer service. Such systems can significantly reduce order turnaround time. They can provide information that enables order clerks to locate items that are out of stock in their own locations. They can help inventory managers balance inventory levels across many warehouses, and they can make information instantly available for answering customer inquiries about the entire range of customer service activities. Exhibit 7 depicts a distributed computing orientation to processing customer service activities in a geographically dispersed organization.

Manufacturing Systems. Manufacturing applications (also known as production applications) include all computer systems used to assist in planning and controlling the manufacturing function of an organization. These include systems that determine raw material and work-in-process inventory requirements, help plan production schedules, aid dispatching operations, and monitor work-order status.

Because the manufacturing process in many organizations is highly complex, involving large numbers of people in many departments, manufacturing computer applications are also necessarily complex. In general, they are characterized by a high degree of subsystem integration, making the tasks of both developing and modifying such systems much more difficult than in other administrative and business areas. As a result, these systems are often very costly and may require years to develop.

In addition, there is normally a high degree of risk associated with manufacturing systems. Their success is dependent on their acceptance by the plant workers who must provide the data for the systems and use the system output. Problems can, and frequently do, arise from employees' lack of understanding of the system or lack of appreciation for the necessity of providing the computer with accurate, complete, timely information.

On the other hand, a very high potential benefit is generally associated with manufacturing systems. Since the manufacturing operation often accounts for the largest portion of product cost, even small improvements in its utilization of resources can generate large profits. Reduction of work-in-process

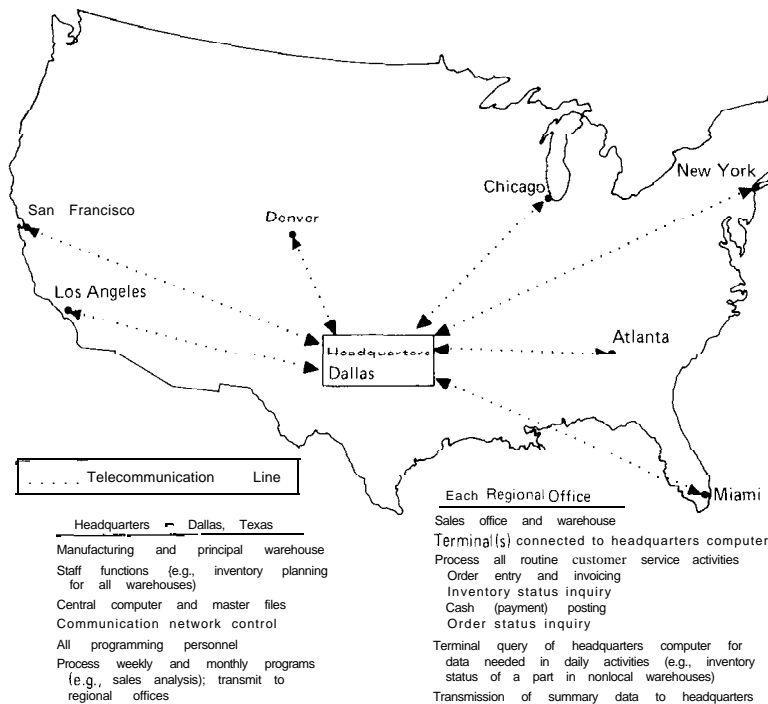


Exhibit 7. Distributed computing-customer service.

inventory, elimination of duplicate parts, shortening of production-planning cycles, and order status information that permits the identification and expediting of behind-schedule orders are some of the benefits that are gained from computerized manufacturing systems.

Exhibit 8 describes the kinds of manufacturing applications used in systems environments of increasing complexity. A prerequisite for manufacturing application systems is an item-coding system that can be used for all subsystems of inventory, bills of materials, engineering changes, and purchasing records. At the simplest stage, inventory accounting systems are implemented, followed by requirements generation (bill of materials explosion) and physical inventory systems.

Up to this point, manufacturing applications are normally used to assist in staff planning and control operations. The next state in the evolution of manufacturing systems, that of preparing work orders used to dispatch jobs, is the first time that the system directly affects the plant worker. Work center analysis helps identify bottlenecks in manufacturing by preparing reports for each work center that show production (actual versus standard), downtime, and backlog.

At the most complex level, the manufacturing computer system calculates work-order schedules and, via terminals located on the plant floor, records

the completion of one job, dispatches the next, and notifies management of all behind-schedule orders.

Accounting and Finance Systems.


Historically, operations such as accounts payable, cost accounting, and financial statement preparation were among the first to be automated, via punched-card systems in the 1930s and 1940s and on computers in the 1950s and 1960s. As in payroll procedures, these functions are largely routine, making them relatively easy to automate. The benefits offered by such applications traditionally have been in the form of clerical cost reduction, and although they have not normally led to dramatic increases in profitability, they have produced sufficiently large dollar savings to more than justify their computerization.

In spite of the traditional nature of accounting and finance applications, several sophisticated, high-benefit systems are being implemented in this administrative and business area. Cash management, financial modeling, and advanced purchasing systems are three examples of the new interest in accounting and finance applications.

In large, diversified, or widely dispersed organizations, the management of cash resources is a difficult job, yet one that has a large impact on profitability. Failure to invest temporarily available cash, premature payment of obligations, or short-

<u>MANUFACTURING SYSTEMS</u>		
Application	Least Complex	Most Complex
	<p style="text-align: center;"><i>On-Line Not Essential</i></p> <p style="text-align: center;">←————→</p>	<p style="text-align: center;"><i>On Line Essential</i></p> <p style="text-align: center;">————→</p>
Production Planning and Control	<ul style="list-style-type: none"> — Work order preparation (manual dispatch) — Engineering specification file maintenance — Work order status reporting 	<ul style="list-style-type: none"> — Work center analysis — Maintenance scheduling — Infinite capacity machine loading — Work order scheduling — Automatic work order dispatching — Exception status reporting (for expediting of work orders, tooling)
Inventory	<ul style="list-style-type: none"> — Weekly stock status reporting with daily activity listings — Item catalog preparation — ABC analysis (distribution by value) — Net return analysis 	<ul style="list-style-type: none"> — “Where used” reporting — Requirements reporting (low-level bill of materials explosion) — Time series requirements reporting — Listing of procurement requirements — Cycle count physical inventory accounting — Requirements forecasting — EOQ calculation — Continuous updating of inventory — Automatic item status reporting — Automatic replenishment initiation (purchase orders, work orders)

Exhibit 8

<u>ACCOUNTING – FINANCE SYSTEMS</u>				
Appl icatio n	Least Complex  Most Complex			
General Accounting	Cost record keeping	Cost accounting • comparison to standards or projected amounts Budgetary accounting Daily exception reporting		Cost estimating
Accounts Payable • A/P	<ul style="list-style-type: none"> ■ Preparation of A/P registers — Check processing ■ Check reconciliation ■ A/P distribution 			
Purchasing		<ul style="list-style-type: none"> — Vendor analysis • volume of purchases ■ Purchase order preparation and follow-up 	<ul style="list-style-type: none"> — Vendor analysis • quality, reliability, price, etc. Derivation of economic purchase quantities (EPQs) 	<ul style="list-style-type: none"> ■ Make-or- buy analysis
Finance		<ul style="list-style-type: none"> ■ Financial statements preparation 	<ul style="list-style-type: none"> ■ Requirements planning • e.g., cash management system ■ Maintenance of stockholder records 	<ul style="list-style-type: none"> ■ Analysis of financial proposals

Note: None of the application systems listed requires on-line or distributed computing capability.

Exhibit 9

ADMINISTRATIVE-BUSINESS APPLICATIONS

sighted investment programs that force an organization to borrow at high rates, all result in less than optimal use of financial resources. In an attempt to avoid these problems, many large organizations are using their computers to help collect, analyze, and report data about cash requirements and reserves. The benefits of such systems include less frequent and smaller short-term loans (a result of being able to project needs over longer periods of time), lower rate loans (a result of being able to forecast needs and investigate multiple sources of debt financing), and higher returns on short-term investments (a result of being able to project how long funds should remain invested).

Financial analysis of proposals is a second area where computers are playing an increasingly important role. Relatively straightforward simulation models enable the financial manager or analyst to generate pro forma statements that show the financial impact of different proposals, such as adding a new product to a current line or opening a new warehouse. The advantage of such models is that many possible outcomes can be evaluated in the time that one or two could be calculated by hand.

Advanced purchasing systems are a third area of current interest. Computer-based systems monitor data about price and quantity discounts, product quality (from product acceptance statistics) and reliability, and speed of delivery. On the basis of these data, computer programs calculate vendor rankings and economic-purchase quantities (EPQs) for individual items and store product information that helps buyers evaluate vendor performance and negotiate favorable contracts with suppliers.

Exhibit 9 lists many of the traditional accounting and finance applications as well as those of high current interest. In addition to those already described, make-or-buy analysis applications in manufacturing companies and computer-based systems for cost estimating are indicative of the trend toward more complex but also higher benefit applications in the accounting and finance area.

Payroll and Personnel Systems. Payroll and personnel systems constitute a fourth important category of administrative and business applications. This category includes all applications dealing in some way with the management and costs of an organization's human resources. The direct benefits of such systems are often less significant (in tangible dollar terms) than other categories of administrative and business systems.


Historically, the payroll function was often the first administrative operation to be computerized.

The highly repetitive nature of the job, with its well-defined rules of computation, made it a logical target for automation, particularly since large numbers of clerical personnel—and hence high clerical costs—were required to perform the function manually. Moreover, by the early 1950s the payroll function in many instances had already been mechanized on punched-card equipment; thus, once an organization installed a computer, it was a relatively simple matter to convert the existing payroll system to the computer (with magnetic tape storage for data files rather than the trays of punched cards used with the former system).

In general, payroll applications have undergone no major changes over the past several years other than in the type of equipment used (e.g., from magnetic tape to disk files for storing data records). An example of one of the few relatively new applications is a computer-based system that simulates the effect of proposed changes in compensation packages. For example, such systems can make it much easier for an organization to evaluate the cost of proposed programs, such as granting a 6.5% across-the-board salary increase or giving an extra holiday to all employees.

In contrast to the relatively few advances in payroll applications, a number of new application systems have been developed in personnel management. In an attempt to cope with rising demands for various kinds of information about employees, many organizations have created computerized personnel systems. These systems are designed to answer requests for information from persons within the organization (e.g., how many employees have not taken their annual vacation this year?) and from outside agencies (e.g., what is the percentage of minority employees in clerical, supervisory, management, and executive positions?). Organizations are also using their personnel data files to monitor and evaluate the effectiveness of personnel management practices. For example, in cases where an organization has a "pay for performance" philosophy, computers help monitor the program by printing out correlations of performance rating and position within salary range. Other programs check to see if all employees are receiving their performance appraisals at the intervals prescribed by the organization.

In organizations with large numbers of employees or wide-ranging skill requirements, data banks containing information about employees' skills and experience have been computerized. When a specific skill mix is needed for an unfilled position, the computer performs a skills search to identify

<u>PAYROLL – PERSONNEL SYSTEMS</u>				
Application	Least Complex  Most Complex			
Payroll	<ul style="list-style-type: none"> Calculation of net payroll from manually calculated gross Preparation of payroll register Check processing Production of required reports (FICA, etc.) 	<ul style="list-style-type: none"> – Attendance accounting – Calculation of net payroll from “hours worked” – Preparation of labor distribution reports 	<ul style="list-style-type: none"> – Evaluation of proposed changes in compensation package 	
Personnel		<ul style="list-style-type: none"> Automation of basic personnel file – Preparation of scheduled reports, e.g., <ul style="list-style-type: none"> • Seniority • Vacation 	<ul style="list-style-type: none"> Salary analysis – Preparation of performance appraisal notices – High-potential employee tracking system Preparation of unscheduled reports (using a generalized information retrieval system) 	<ul style="list-style-type: none"> – Skills inventory accounting (skills search) – Manpower planning analyses * – On-line payroll/personnel record maintenance at location of employee (e.g., plant)

Note: Other than the system identified with the asterisk, none of the application systems requires on-line or distributed computing capability

Exhibit 10

ADVANCED RESEARCH PROJECTS AGENCY

(and print out the names of) those who meet the qualifications. These systems are often very costly to establish and keep up to date, and some organizations have discontinued them after finding that their use frequency did not justify their expense.

Exhibit 10 describes the range of payroll and personnel applications systems that are found today in industrial, commercial, and governmental organizations.

REFERENCES

1966. Dearden, John. *Computers in Business Management*. Homewood, Ill.: Dow Jones-Irwin. (Hardware and software fundamentals, management problems of computer systems, mathematical programming, and future impact of computers in management.)
1968. Boutell, Wayne S. *Computer-Oriented Business Systems*. Englewood Cliffs, N.J.: Prentice-Hall. (Basic forms of application systems (e.g., batch), data processing department organization, introduction to hardware and software.)
1968. Heany, D. F. *Development of Information Systems*. New York: Ronald Press. (History of computer use in business data processing; projections about future use; begin at page 373.)
1969. Orlicky, Joseph. *The Successful Computer System*. New York: McGraw-Hill. (Application systems development.)
1970. Humphrey, Sturt, and Ronald Yearsley. *Computers for Management*. New York: American Elsevier. (Computer applications in business; e.g., marketing.)
1970. Krauss, Leonard I. *Computer-Based Management Information Systems*. New York: American Management Association. (Management-oriented treatment of concept, development, and implementation of management information system.)
1971. Smith, Leighton F. *An Executive Briefing on the Control of Computers*. Park Ridge, Ill.: Data Processing Management Association. (Development and management of computer systems -analogy to factory operation.)

G. GLASER AND A. L. TORRANCE

AIKEN, HOWARD

For articles on related subjects see **DIGITAL COMPUTERS**, Early; **MARK I**; and **WATSON, THOMAS, SR.**

Howard Hathaway Aiken was born March 8, 1900, in Hoboken, N.J., and died March 14, 1973, in St. Louis, Missouri. He grew up in Indianapolis, Indiana, where he attended Arsenal Technical High School while working 12 hours a night at the Indianapolis Light and Heat Company. Upon graduation he went to work for the Madison (Wisconsin) Gas Company, a position that allowed him to go to the University of Wisconsin. He received his B.A. degree in 1923 and was immediately promoted to chief engineer at Madison Gas.

In 1935 he returned to school, first at the University of Chicago and then at Harvard. His doctoral thesis at Harvard, resulting in a Ph.D. in 1939, was on the theory of space charge conduction. The research required laborious calculations of nonlinear differential equations. This experience led him to investigate the possibility of performing these types of calculations with machine assistance. His thoughts on this subject led him in 1937 to circulate a memo entitled, "Proposed Automatic Calculating Machine" (reprinted in *Spectrum*, August 1964, pp. 62-69).

Harvard was not the most likely environment to get support for this type of research. Fortunately, Harvard professors Ted Brown (Business) and Harlow Shapley (Astronomy) were impressed with his work, and both knew of the interest of Thomas Watson Sr. in projects of this nature. With their encouragement, and the knowledge that IBM had the necessary technology, Aiken approached Watson. A contract was signed in 1939 whereby IBM, with financial support from the U.S. Navy, would build the Automatic Sequence Controlled Calculator (Harvard Mark I). The machine was running in 1944, and Aiken and Grace Hopper described it in a paper in *Electrical Engineering* (Vol. 65, 1946, pp. 384-391, 449-454, 522-528).

The Mark I was followed by the Mark II (a relay machine built for the Naval Proving Ground at Dahlgren and completed in 1946), the Mark III (an electronic machine, also for Dahlgren, completed in 1950), and the Mark IV (an electronic machine built for and delivered to the Air Force in 1952). With the completion of Mark IV, Aiken got out of the business of building computers.

ADVANCED RESEARCH PROJECTS AGENCY. See **ARPA NETWORK.**



Fig. 1. Howard Aiken

It is difficult to evaluate precisely the impact of Aiken's series of machines and the Harvard Computation Laboratory which he founded. Fortunately, the documents are available to anyone interested. One need only look at the log books of the computation lab for this period to see the worldwide range of people who visited the laboratory. Another source of Aiken's work is the many publications in the "Annals of the Harvard Computation Laboratory" series. The Harvard catalog also provides clear evidence of the existence of courses in "computer science" a decade before the emergence of this program at most universities.

In 1947 and again in 1949 Aiken organized symposia on large-scale digital devices at Harvard. Programs from both meetings strongly reflect his hand and his philosophy at that time. Perhaps his most profound impact was in the environment he created at Harvard, which enabled the University to become a vital training ground for many people who are outstanding in the field today. A perusal of those who did their doctoral dissertations under his direction is an excellent example of this impact.

Aiken retired from Harvard in 1961 and moved to Fort Lauderdale, Florida, where he formed Aiken Industries. He also joined the faculty of the University of Miami as Distinguished Professor of

Information Technology. In this latter position, he helped the University develop a computer science program and design a computing center.

His honors are much too numerous to mention in detail. They include honorary degrees (University of Wisconsin, Wayne State University, and Technische Hochschule, Darmstadt), prizes (Rochlitz Prize, Edison Medal of IEEE, the John Price Award of the Franklin Institute) as well as medals from both the United States (Air Force and Navy for distinguished service) and foreign governments (Sweden, Belgium, France, and Spain).

Howard Aiken felt that he had to be continuously involved in challenging endeavors in order to stay alive both physically and intellectually. His career is a document of that creed. Some of his detractors accused him of living in the past, but nothing could be further from the truth. He was a man of rare vision, whose insights have had a profound effect on the entire computing profession.

REFERENCES

1947. Anon. "Howard Hathaway Aiken," *Current Biography*, pp. 5-7.
 1973. Oettinger, Anthony G. "Howard Aiken," *Communications of the ACM*, May, pp. 298-299.

H. S. TROPP

ALGEBRA, BOOLEAN. See **BOOLEAN ALGEBRA.**

ALGEBRAIC MANIPULATION LANGUAGES

For articles on related subjects see **ARTIFICIAL INTELLIGENCE; LIST PROCESSING LANGUAGES; NUMERICAL ANALYSIS; and SYMBOL MANIPULATION.**

For articles on related terms see **PORTABILITY; and TREE.**

Algebraic manipulation languages comprise a family dedicated to a single application area, the symbolic computations of applied mathematical analysis. A number of names have been suggested and employed to refer to this field. Among the most common are Symbol Manipulation, Formula Ma-

Simplification. The theoretical situation here is a bit muddy. We are, of course, on firm ground as long as our system is dedicated to the manipulation of a class of mathematical objects; e.g., polynomials, rational functions or truncated power, Fourier or Poisson series, which admit a canonical representation. Many systems are so dedicated while others often include rather self-contained packages for these seminumerical computations. A **typical** tiny example of the automatic simplification facilities offered by such systems would be the transformation of the expression.

$$\frac{A(X + A)/(1 - x * A)^2 + 1/(1 - x * A)}{1 + [(X + A)^2/(1 - X * A)^2]}$$

into

$$\frac{1}{X^2 + 1}$$

This is not to suggest that these systems have no problems in giving the user sufficient control over the form in which his answers are presented, in choosing and implementing efficient representations and algorithms-this is, in fact, probably the most active and contentious research area in algebraic manipulation today-or in avoiding the practical pitfalls [e.g., the forcing of the binomial expansion of $(x + y)^{1000}$] associated with too heavy reliance on canonical forms. We are only emphasizing the facts that as long as the domain possesses a canonical form, it can always be decided by very simple means whether two of its objects are equivalent, and that as we pass to more complex domains, the equivalence of two expressions may be undecidable by any means.

Most of the general-purpose simplification programs that have been implemented to date rely on some well-ordering of the admissible expressions, which allows for rapid checks for combinations or cancellations in complex sums or products, provided the equivalences can be recognized. Such an algorithm is usually augmented by a collection of disparate facts, such as $\cos(0) = 1$ or $e^{\log(x)} = x$, which are to be applied as local transformations. Applied recursively, such rules can easily find such simplifications as the reduction to zero of

$$y[\exp(\log(\cos(0))x - t \sin(0))] \sim xy.$$

What capacity for global transformations (e.g., simplification via the familiar trigonometric identities) might exist is usually in the form of allowance for

user-defined transformations to be applied interpretively and disastrously inefficiently.

While there have been a number of interesting though isolated theoretical results on the reduction of certain wider classes of algebraic expressions to canonical forms, they have had remarkably little effect on the design of systems for the simplification of broad spectra of mathematical expressions. A few partial exceptions to this are the work of Moses (1969) on the implementation of algorithms due to Risch for the integration of certain classes of elementary functions, the **RADCAN** facility of MACSYMA, and some simplification algorithms recently implemented within REDUCE (see later section "Systems"). Typical of such transformations would be the replacement of

$$\frac{\log(A^{2X} + 2A^X + 1)}{\log(A^X + 1)}$$

by 2.

Integration. Probably no problem in algebraic manipulation has excited more lay interest and is in more demand during demonstrations than indefinite integration. We will discuss only the history of indefinite integration of elementary functions and defer on both definite integration and the **more** general question of the integration of differential equations, even though some programs exist for these problems.

The first program written to attack the problem of symbolic integration was Slagle's thesis, SAINT. To this day, this program is the purest example of the application of classical tree-pruning, backtracking, artificial intelligence techniques to algebraic manipulation. It was an awesome achievement, especially considering its date-1961; and it contained features, especially its semantic pattern-matching facility, which addressed areas that remain difficult today.

None **theless**, the basic artificial intelligence approach was soon to be abandoned as the path to symbolic integration. The next relevant program was that of Manove et al., (1968), written as part of the **MATHLAB** system for the integration of rational functions, a well-defined domain of integrands for which an algorithmic attack was available. This program was also important in that it contained the first complete implementation of an algorithm for the factorization of multivariate polynomials **over** the integers. It led, too, to an interesting algorithm for the inverse **Laplace** transform of rational functions, also implemented for **MATHLAB** and sharing

a great deal of code with the integration algorithm. This program dates from 1964 to 1966.

By the end of 1967, Moses had completed the thesis version of his symbolic integration program, SIN. This program far excelled SAINT in both breadth of success and speed of solution. It comprised a three-step sequential attack:

1. Determine if the integrand can be expressed as a constant multiple of an expression of the form $f(u(x))u'(x)$, where f is a member of an extremely narrow class of functions whose antiderivatives are known. If so, the solution is immediate.

2. If the integrand cannot be so expressed, the program passes into a "bag of tricks" stage similar to SAINT except that no backtracking ever takes place. The program is able, employing a semantic pattern-matching facility (again like SAINT), to classify each problem into *one* (quite unlike SAINT) of a finite number (eleven) of categories; e.g., trigonometric, rational function, rational function of logarithms, etc. For each such category, the program continues the analysis, finally settling on exactly one trick, which might or might not work. While still heuristic, this second stage really contained a great deal of practical knowledge about integration. It employed as a subroutine, and was deeply dependent upon, the rational function integration program of Manove, Bloom, and Engelman. For example, if SIN were asked to integrate $\int x^2 \sin^{-1}(x) dx$, it would classify this problem as "Method 9, rational times arctrigonometric" and perform an integration by parts, yielding

$$\frac{x^3 \sin^{-1}(x)}{3} - \frac{1}{23} \int \frac{x^3 dx}{1 - x^2}.$$

This would bring it to

$$\int \frac{x^3 dx}{(1 - x^2)^{1/2}}.$$

SIN would classify this problem as "Method 4, binomial-Chebyshev" and substitute y for x^2 , yielding

$$\frac{1}{2} \int \frac{y dy}{(1 - y)^{1/2}},$$

followed immediately by the substitution of z^2 for $1 - y$, yielding

$$- \int (1 - z^2) dz.$$

The problem is now $\int (1 - z^2) dz$. This is a polynomial, a trivial case of a rational function. The next call, this time to the **MATLAB** package (Method S), is the last, yielding

$$\int (1 - z^2) dz = z - \frac{z^3}{3}.$$

Finally, SIN must unwind its stack by a series of inverse substitutions, which yield the final answer:

$$\begin{aligned} \int x^2 \sin^{-1}(x) dx &= \frac{1}{3} x^3 \sin^{-1}(x) \\ &+ \frac{1}{3} (1 - x^2)^{1/2} - \frac{1}{6} (1 - x^2)^{3/2}. \end{aligned}$$

3. The third stage (entered only when the first two have failed) contained, in addition to integration by parts, a powerful heuristic method, called **EDGE** (for **EDUCATED GUESS**) based on a reasonable guess as to the possible form of the integral. It was, in some ways, a precursor to the algorithmic solution to the problem of integration of elementary functions, soon to be discovered by Risch in the 1968-1970 period.

The Risch method represents the final passage of the problem of symbolic integration from heuristic methods to algorithmic ones. It is, in its greatest generality, dependent on results from algebraic geometry and is certainly beyond any reasonable discussion here. We should like to point out, though, that it probably represents the most deeply recursive application known of the idea of "undetermined coefficients." Some parts have been implemented by Moses, but at this time the general case defies our capability to analyze its complexity.

Systems. We prefer the word "system" to "language" partly because the concern here, as with most application-oriented situations, tends not to be with the questions most frequently in the fore during discussions of general-purpose computer languages (e.g., syntax, control mechanisms, variable bindings, macrofacilities), but more on the semantics of the computational machinery provided. Can it integrate? What functions? Definite and indefinite? Facilities of this sort, by necessity, accrete to form a lumpy porridge that no self-respecting "language" designer would acknowledge. At times, the only reasonable way to determine the usefulness of such a facility with respect to a given problem is by experimentation. Another reason for our preference of the word "system" is that so many of the extralinguistic features, the display programs, the file management,

ALGEBRAIC MANIPULATION LANGUAGES

the editors, and the libraries are so applications oriented.

We will soon list some of the major systems along with capsule descriptions that can serve only as a source of first impressions for those who might be shopping. Anyone seriously interested should, as a more meaningful introduction to the field, obtain a copy of the Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation (Petrick, 1971), which is still remarkably *au courant*. We will use it as a universal reference for contemporary systems, both for description of facilities and as a source of contacts from whom the various systems may be obtained. In reference to this last point, the reader should understand that, while the various systems vary enormously as to their portability, it is the desire of most of the authors to provide the widest possible user community for their programs and therefore most systems are available (usually source code as well as binary program) to all. In those few cases where our understanding is otherwise, we will note it below.

What questions would we expect a user to ask before choosing a system? While they must obviously depend on the anticipated application, the following certainly must be among the most common :

1. Is it available to me? Is it well documented? How much effort is involved in my setting it up and learning to use it?

2. What data types are available? Does it provide variable precision integer (rational) arithmetic? Does it admit only, say, polynomials or can it simplify general mathematical expressions? Differential equations? Matrices? Tensors?

3. What transformations are delivered with the system? Are arithmetic, substitution, simplification, and differentiation sufficient for my problems, or do I need more advanced features such as polynomial factorization, matrix inversion, integration, or Laplace transforms?

4. How efficient is the system for my problem domain? Am I likely to lose because the intermediate expressions swell to the point of destroying the computation? Would another system be much faster or fit better within the space allotted by my facility?

5. Can I edit my expressions? Can I easily obtain equivalent but more revelatory forms for the answers? Who is boss?

6. Will the system provide legible two-dimensional displays of my expressions? Does it operate interactively so that I can decide what to do next on the basis of what has happened? Does it provide

good file management for the support of continuing computation?

7. Does the system come with a useful library of symbolic routines? Can the formulas generally be compiled as numerical programs?

It would be too confusing to list the systems in some arbitrary (e.g., chronological) order. To introduce a little sense into the listing, we will group them according to a single characteristic: their internal data representations. We mean this only in the broadest possible sense; i.e., while we will ask whether a system is, say, restricted to canonically represented polynomials, we will not go into the question of which canonical representation. Our choice of criterion is quite likely biased by our experience as systems designers, but it is extremely important from a user's viewpoint, too, and in fact is probably the most decisive single factor in determining what the system can and cannot do for him. We will establish four categories and place each system within one. In some cases the choices are borderline, but no real harm can ensue should our choice be different from the one the system's authors would have made. The four categories are discussed below.

Seminumerical. The name "seminumerical" is borrowed from Knuth. These systems are constructed to manipulate data from rigid classes of mathematical objects possessing strictly canonical forms. The classes most generally treated are multivariate polynomials, rational functions, and truncated power series. However, sometimes other classes, such as Laurent series or trigonometric series, are accepted. The operations performed by these systems are generally closed with respect to the domain. So, for example, one should expect a rational function system to differentiate—since the derivative of a rational function is rational—but not, in general, to integrate or perform inverse Laplace transforms, since these lead to logarithmic and exponential results, respectively. These systems have very little to do with "symbolic" computation and are in fact much closer to being very powerful arithmetic facilities. Hence the name.

Ghost. These systems appear externally to manipulate quite general mathematical expressions, but appear internally to be functioning with canonically represented data, much like the seminumerical systems. When their services are sufficient, they can represent quite intelligent compromises, providing limited but often adequate manipulation of general expressions with the efficiency normally associated with seminumerical systems.

Symbolic. These systems admit the most general species of mathematical expressions, usually representing them as quite general tree structures. While it is precisely this generality that allows for the provision of such features as user-defined pattern replacements, this characteristic often makes the system inadequate for performing certain tasks because it fails to provide some quite commonly required special machinery, such as a reasonably efficient algorithm for the greatest common divisor of two polynomials. This particular lacuna is in fact the reason that purely "symbolic" simplifiers are so poor at division.

Hybrid. These are generally the most ambitious systems, accepting the broadest spectrum of mathematical expressions in general but possessing, in addition, special representations and special algorithms for particular special classes of expressions. Typical of a hybrid is a symbolic system that possesses, for use when appropriate, a special package for the manipulation of multivariate rational functions over the integers. An excellent example of the potential of such systems is the **MATHLAB** program for the automatic solution of linear differential equations with constant coefficients. The method of solution involves the computation of both direct and inverse Laplace transforms. It is approximately true that the general symbolic features are used for the initial pattern-matching problem classification, the canonical rational function package for the inverse Laplace transforms, and both for the direct transforms.

SEMINUMERICAL SYSTEMS. We will omit discussion of, as probably outside the interests of our general reader, a number of very special purpose systems that fall within this category. We feel compelled, however, to mention the system of Deprit, Henrard, and Rom, which is constructed basically for a single computation—the reproduction and extension of the **DeLaunay** lunar theory expansion. This splendid achievement probably represents a mechanical computation on an order of magnitude greater in size than any achieved by earlier researchers.

ALTRAN (Petrick, 1971; pp. 153-157). Developed at Bell Labs out of its predecessor **ALPAK** system, **ALTRAN** is perhaps the most typical of seminumerical systems. It provides precisely for the arithmetic (through greatest common divisors, but not factorization) of multivariate rational functions over the integers and truncated power series. It is written in **Fortran**, but a few primitives must be hand coded. Its authors see this as a method of achieving portability while making no untoward sacrifice of efficiency.

SAC-I (Petrick, 1971; pp. 144-152). This is a large system of **Fortran** subroutines, callable from **Fortran**, for the manipulation of multivariate rational functions with (infinite precision) integer coefficients. This implementation does have some unfortunate consequences for **SAC-I** programming in that, unlike most other systems that create an applications-oriented environment, the burden of **Fortran** accounting is on the user. The excerpt in

```

IMPLICIT INTEGER (A-Z)
LOGICAL LASTIN
DIMENSION Z(30), ZZ(30), ZPWR(30, 3), DZ(30), DDZ(30),
1  ABCD(4), PARTL(15)
COMMON /TR1/ AVAIL, STAK, RECORD(72)
1  /TR2/ SYMLST
COMMON /PMODE/ KMPRES
COMMON /EPSLON/ ECODE, EPSNAM, EPSLST, ELAST, ETAT L
COMMON /PROBLM/ ZPWR, DZ, DDZ, PARTL, ZAZB, DZADZ28, ZDOZAD, DZDZ3
1  , PFOOCT, SOTOTL, OTHERZ, OTHER1, DENOM, LIMIT, LVDUMP, N, ABCD
2  , MAXLTM, L1, L2, L3, L4, L5, L8, L12, L24, L48, LM4, LM12, LM24
COMMON /EXTRA/ RESERV
EQUIVALENCE (ZPWR(1,1),Z(1)), (ZPWR(1,2),ZZ(1)),
1  (ZPWR(1,3),ZZ(1)),
2  (ABCD(1),A), (ABCD(2),B), (ABCD(3),C), (ABCD(4),D),
3  (PARTL(15),P4L2), (PARTL(4),P4L8), (PARTL(8),P4L12),
4  (PARTL(3),P4L24), (PARTL(10),P4L48),
5  (PARTL(12),P3L1), (PARTL(9),P3L4), (PARTL(1),P3L12),
6  PARTL(6),P3L24)
DATA INPUT, OUTPUT /5, 6/
CALL CATCHR(-1)

```

Fig. 3

ALGEBRAIC MANIPULATION LANGUAGES

Fig. 3 from a SAC-I program exhibits this unnaturalness. Note that the users' concern is as much with type declarations and space allocations as with the mathematics. Algorithms supplied include those for rational function arithmetic, polynomial factorization, the solution of simultaneous linear equations, and one that returns (only) the rational part of the integral of a rational function.

This system is important, not only for itself but also for the focus it has provided for the prolific school centered around G. E. Collins at the University of Wisconsin for the study of seminumeric algorithms. A series of Ph.D. dissertations has produced not only many new algorithms but also the majority of the worst-case analyses of such algorithms that we possess today.

CAMAL (Petrick, pp. 134-143). On the borderline of being classified as a ghost system, this highly efficient physics-oriented system is probably not exportable beyond the Titan machine in Cambridge. It contains seminumeric modules for the arithmetic of polynomials (not rational functions), truncated power series, and trigonometric series with polynomial coefficients. It differs from almost all other systems we will mention in that the burden for the "garbage collection" of the space occupied by those intermediate expressions that are no longer needed in the execution of a program is placed on the user rather than being supplied automatically by the system. CAMAL has been credited with a number of computational successes, particularly in celestial mechanics and general relativity.

SYMBAL (Engeli, 1970). Restricted to polynomials, rational functions, and truncated power series, this system is distinguished primarily by its elegant Algol-like syntax. Engeli provides many extremely concise programming examples.

Fig. 4 exhibits a sample of the code and a few lines of output for the computation of the Legendre polynomials.

GHOST SYSTEMS

REDUCE 2 (Petrick, 1971; pp. 128-133). Starting from a canonical form suited to multivariate polynomials, REDUCE 2 extends itself by a variety of means to the manipulation of quite general mathematical expressions, relying on the polynomial procedures for basic simplification. It is quite different from seminumeric programs in its support of a (rather constrained) user-defined pattern replacement facility and of a 1'-dimensional (exponents raised, but no nice two-dimensional (2-D) fractions like those in Fig. 2) mathematical display program.

A REDUCE program for computing the *F* and *G* series mentioned at the beginning of this article would look like Fig. 5.

The REDUCE 2 system supplies, in addition to its general-purpose routines, a significant facility specialized to the multilinear algebra associated with high-energy physics. This part of the system has proved extremely successful, with numerous published physics papers citing it as the computational vehicle.

IAM (Petrick, 1971; pp. 115-127). This is a remarkably ambitious system, considering its basic dependence on a single canonical data representation. Included are a number of advanced facilities generally associated with hybrid systems (e.g., polynomial factorization), a SIN-like indefinite integration program, and good 2-D output. This is a proprietary program; i.e., the binary program is available for a fee; the source code is not. The system is receiving minimal support at this time.

begin

```
P:= {0:1, x, 25:};
for I:= 2:25 do
    P[I] := (2*I-1)/I*x*P[I-1] - (I-1)/I*P[I-2];
```

end

```
P := {0:25:};
P[2] := - 1/2 + 3/2 * x ^ 2;
P[3] := - 3/2 * x - 5/2 * x ^ 3;
```

```
P[25] := 16900975/4194304*x - . . .
```

Fig. 4

```

DEPS← -SIG*(MU+2*EPS)$
DMU←-3*MU*SIG$
DSIG←EPS-2*SIG+2$
F←1$
G←0$
FOR I←1 STEP 1 UNTIL 12 DO
BEGIN
    F1←-MU*G + DEPS*DF(F,EPS)+DMU*DF(F,MU)+ DSIG*DF(F,SIG)$
    WRITE "F(",I,"")←",F1;
    G1←F + DEPS*DF(G,EPS)+ DMU*DF(G,MU)+ DSIG*DF(G,SIG)$
    WRITE "G(",I,"")←",G1;
    F←F1$
    G←G1$
END;
    
```

Fig. 5

```

*TYPE EQA; TYPE EQB; TYPE EQC

EQA:  A = 
$$\frac{12 A L + 15 B L + 20 C L + 20}{60}$$


EQB:  B = 
$$\frac{10 A L + 12 B L + 15 C L + 15}{60}$$


EQC:  C = 
$$\frac{30 A L + 35 B L + 42 C L + 42}{210}$$


*SOLVE EQA FOR B
*ELIMINATE B FROM EQB
*ELIMINATE B FROM EQC
*SOLVE EQB FOR C
*ELIMINATE C FROM EQC
*SOLVE EQC FOR A

*TYPE A

A: 
$$\frac{-1575 L + 126000}{3 L^2 - 4140 L - 226800 L + 378000}$$


*ELIMINATE A FROM EQA
*ELIMINATE A FROM EQB
*SOLVE EQA FOR C
*ELIMINATE C FROM EQB
*SOLVE EQB FOR B
*TYPE B

B: 
$$\frac{2100 L + 94500}{3 L^2 - 4140 L - 226800 L + 373000}$$


*ELIMINATE B FROM EQA
*SOLVE EQA FOR C
*TYPE C

C: 
$$\frac{-L^2 + 3510 L + 75600}{3 L^3 - 4140 L^2 - 226800 L + 378000}$$

    
```

Fig. 6

As an example of the conceptual level of this system, we present in Fig. 6 the conversation accompanying the symbolic solution of three simultaneous linear equations.

ALA DIN (Petrick, 1971; pp. 90-99). This system uses the **MATHLAB** rational function package (Manove et al.) for simplification. Its primary contribution is its display program, which employs the graphical capabilities of the IBM 2250 for the creation of high-quality (approaching textlike) two-dimensional presentations of mathematical expressions. It also supports the use of a **lightpen** for subexpression selection.

SYMBOLIC SYSTEMS. We should mention first a sequence of three early programs for the simplification of general symbolic mathematical expressions represented as prefix-notation tree structures. The first, at M.I.T., was due to Mart, and the other two were due to Wooldridge and Korsvold at Stanford. The latter has survived in current usage as a result of its incorporation, subject to modification, into the **MATHLAB**, **MACSYMA**, and **SCRATCHPAD** systems.

In the mid- 1960s there appeared two systems, **Formula Algol** and **FAMOUS**, which, while dedicated to the symbolic manipulation of mathematical expressions, presented the user with almost no built-in automatic simplification facilities. This was due, at least in the case of **FAMOUS**, to a conscious decision that, since the "simplicity" of an expression is surely context-dependent, it should be reasonable to present the user with *complete* control over the simplification process. That is, the user should be compelled to define all transformations, rather than, as with most systems, be permitted simply to switch on and off the transformations supplied by the system architects. No system of this species has ever solved the inherent efficiency problems to the extent

ALGEBRAIC MANIPULATION LANGUAGES

that it could serve more than didactic purposes. Probably neither Formula Algol nor FAMOUS could be revived today.

Another lost symbolic system of importance is the Symbolic Mathematical Laboratory of W. A. Martin. This system provided high-quality 2-D graphics on a DEC-340 display and was also the first to employ a **lightpen** for subexpression selection. In some ways, it represented a degree of interaction that has not been duplicated by any subsequent system. Nor were its innovative internal programming techniques restricted to its graphics facilities. Of particular interest is the use of hash coding for subexpression matching (Petrick, 1971; pp. 305-310).

The best known, purely symbolic systems are, of course, **Formac** and its current version **PL/I-Formac** (Petrick, 1971; pp. 105-114). **Formac** was the first widely available general-purpose algebraic manipulation system and served for a period to define the field. Certainly, there was a time when one could have safely made the statement that the majority of all mechanical symbolic mathematical computations had been done within **Formac**. The practical success of these systems, in spite of their rigidity with respect to user modifications and their lack of any seminumerical facilities for rational function computations, is probably due to the overall intelligence of the facilities that were provided. Above all, they were certainly sufficient to support the dominant application area of truncated power series expansion. Current support is minimal.

HYBRID SYSTEMS

MA THLAB (Petrick, 1971; pp. 29-41). This system is distributed currently for on-line operation on the DEC system-10 (PDP-10) computer, although subsystems have been converted to run on IBM and CDC machines. This was the first heavyweight hybrid system passing data freely between a general-purpose simplification package and a powerful rational function package. Marred by the lack of a number of practical necessities, this system is probably most important for its computational innovations. These include the first complete program for the factorization of multivariate polynomials over the integers, and consequently for the partial fraction expansion of rational functions; for the integration of rational functions; for the inverse Laplace transform of rational functions; for the solution of linear differential equations with constant coefficients; and for the solution of equations via polynomial factorization. In addition it contains **CHARYBDIS**, the first program for the two-dimensional display of mathematical expressions on typewriter-like devices (Teletypes, alphanumeric displays, line-

printer, etc.). Its dedication to an on-line environment led to an interesting command structure and a number of convenient core-oriented and disk-oriented bookkeeping facilities.

As a first example of the facilities of this system, we should like to return to the preceding example, introduced in our discussion of the **IAM** system, of the symbolic solution of simultaneous linear equations. Because of the still higher conceptual level of **MATHLAB**, at least with respect to this problem, the entire conversation would be condensed into the single instruction:

```
'SIMSOLVE ('EQA, 'EOB, 'EQC, A, B, C)$
```

The output would be almost identical.

A further demonstration of the expertise of this system is provided in Fig. 7 by the conversation representing the solution (controlled by the machine, not the user) of the differential equation representing the motion of a velocity-damped spring. The lines starting with “#” are those typed by the user.

MACSYMA (Petrick, 1971; pp. 58-75). In many ways a descendant of **MATHLAB**, this leviathan of the field possesses enough sheer code to approach an order of magnitude dominance over many other systems. It can do just about anything any other system can do, with the obvious exception of certain very specialized capabilities, such as the **REDUCE** high energy physics machinery. Furthermore, considerable attention has been devoted recently to insure that it concedes little in efficiency to less general systems.

It is impossible to summarize here its facilities, ranging as they do from extremely flexible user control over the form in which rational functions are presented to a semantic pattern-matching facility that, at least, if taken together with **SCHATCHEN** (Moses, 1969), serves to define the state of the art. Features such as programs for the manipulation of polynomials over the Gaussian integers or the best extant program for the computation of symbolic limits (Petrick, 458-464) are almost lost in the enormity of this first system to approach the goal of an algebraic manipulation facility.

As an example of the power of **MACSYMA**, we return a third time to the symbolic solution of the simultaneous linear equations. Again, as in the case of **MATHLAB**, a single instruction will suffice:

```
SOLVE ([EQA, EQB, EQC], [A, B, C]).
```

Again, the output is almost identical. What is different here is that the instruction is at a still higher

```

DSS:DERIV(X,T,2)=-K*X-A*DERIV(X,T)$

      2
      D X      D X
      --- = (-K)X  A - - -
      2      DT
      DT

# 'LDESOLVE('DSS,X,T)$
NEED INITIAL CONDITIONS
#ASK$
X(0)
#L$

$$\frac{4K - A^2}{4}$$

#0$

IS THE EXPRESSION

      2
      4K - A
      -----
      4

TO BE CONSIDERED POSITIVE NEGATIVE OR ZERO?
#POSITIVE$

      A
      - - - T
      2
E      (L**COS(1/2**SQRT(4K-A )T) + L**A-----
                                     2
                                     SIN(1/2**SQRT(4K - A )T)
                                     -----
                                     2
                                     SQRT(4K - A )
    
```

Fig. 7

level, since the instruction `SOLVE` is more generic and the program must classify the problem as the solution of simultaneous *linear* equations. Chosen to exemplify the many expert facilities possessed uniquely by this system, the following example is given for the computation of the residue of a meromorphic function. The command is

```
RESIDUE (SIN (A*X)/X**4, X, 0, 4).
```

The answer is $-A^3/6$.

At present the only reasonable route of access to MACSYMA is through the ARPA network (Arpanet). This is due primarily to the complexity of the system, but is also a result of its author's reluctance to distribute source code.

SCRATCHPAD (Petrick, 1971; pp. 42-58). This most eclectic of systems derives most of its considerable computational powers from the direct accretion of code from such sources as (especially) REDUCE, MATHLAB (including CHARYBDIS),

SIN, Korsvold's simplification program, and Martin's graphical display programs.

An important innovation of this system is its highly expressive and succinct syntax, which not only allows the user an especially wide class of natural notational devices but also provides him with the means of extending them himself. As an example of this naturalness, we would like to return to the succinct definition of the Legendre polynomials presented in connection with SYMBAL (see Fig. 4) and now present the SCRATCHPAD version:

```

p<0> = 1
p<1> = x
p(i) = (2*i - 1)/i * x * p<i - 1> - (i - 1)/i * p<i - 2>,
      i in (3,4,...)
i in (0,1,...,25) p(i)
    
```

The last line invokes the actual computation and printing of the results.

ALGOL 68

Another unique feature of this system is, as a result of its commitment to interactive problem solving, the preservation of enough information to allow the user to return not only (as with several systems) to old results, but also to past *states*.

The system is actively in current use only at the IBM T.J. Watson Research Center.

It is important to acknowledge the enormous debt algebraic manipulation owes to the invention and development of the symbolic list processing language Lisp. Among the systems mentioned in this article, those written in Lisp include SAINT, MATHLAB, SIN, REDUCE, ALADIN, the simplification programs of Hart, Wooldridge, and Korsvold MACSYMA, FAMOUS, Martin's Symbolic Laboratory, SCHATCHEN, SCRATCHPAD, and CHARYBDIS.

REFERENCES

1968. Manove, M., S. Bloom, and C. Engelman. "Rational Functions in MATHLAB." In D. G. Bobrow (Ed.), *Symbolic Manipulation Languages and Techniques*. North Holland, Amsterdam.
1969. Moses, J. "Symbolic Integration, MAC-TR-47, Project MAC," M.I.T.
1970. Engeli, Max E. "SYMBAL, Summary + Examples." Zurich: FIDES Union Fiduciaire.
1971. Petrick, S. R. (Ed.). *Proc. Second Symposium Symbolic and Algebraic Manipulation*, Los Angeles. Available from Association for Computing Machinery, New York.
1972. Horowitz, Ellis. "The Application of Symbolic Mathematics to a Singular Perturbation Problem," *Proc. ACM Annual Conf.*, Boston.

C. ENGELMAN

ALGOL 6%

For articles on related subjects see **PASCAL**; **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.

For articles on related terms see **BLOCK STRUCTURE**; and **IDENTIFIER**.

Algol 68 is a language designed by a working group (WG 2.1) of the International Federation for Information Processing (IFIP) in order to provide a

general-purpose programming language that would be suitable for communicating algorithms, executing them efficiently on different computers, and teaching computer science. Even though Algol 68 is a successor of Algol 60, it is a completely new language, different from Algol 60 in many essential aspects. Its design reflects the 1968 understanding of a number of fundamental concepts of programming languages and computer science.

Algol 68 has great expressive power and yet a very elegant and interesting basic structure. It features five primitive types (called "modes") of values: **bool** (boolean), **char** (character), **int** (integer), **real** and **format**; and five rules for constructing new modes from the already defined ones. So, for example, values of mode **[] real** are one-dimensional arrays or *multiple values* of reals. Values of mode **struct ([] char name, bool sex, int age)** are personal records or *structured values*. Values of mode **union (real, int)** are either reals or integers, but no value of this mode can be both of mode **real** and **int**. References are values that refer (point) to other values. For example, values of mode **ref [] char** are references to one-dimensional arrays of characters. Values of mode **proc (int, real) bool** are *routines* (i.e., procedures) that take two arguments of respective modes **int** and **real** and return a value of mode **bool**.

Since references and routines are values, they can be manipulated like any other values. In particular, they can be passed as parameters in procedure calls. Because of this it is possible to achieve the effects of three types of procedure calls found in other programming languages: call by value, call by name, and call by reference. So, for example, values of mode **proc (ref [] char, int) int** are routines with the first formal parameter called by reference.

Different sorts of declarations (for example, array declarations and switch declarations) found in other programming languages are captured in the *identity declaration* of Algol 68. This concept is also the basis of the parameter-passing mechanism; it allows construction of an infinite number of new modes from the already defined ones and permits declaration of arithmetic and logical operators and their priorities.

The identity declaration and the concept of a reference clarify the distinction between a variable and a constant. An identity declaration in a program defines the value possessed by the identifier that appears in the declaration. This value may be a reference to another value, in which case the identifier is declared as a variable. An example of an initialized (i.e., one that includes assignment) declaration of that sort is **real x; = 3.14**, which gives rise

to the Following scheme:

identifier $x \rightarrow$ reference to a real value $\rightarrow 3.14$.

The effect of a standard assignment statement is achieved by making the reference possessed by the identifier refer to the value specified in the statement. This is not possible if the value possessed by an identifier is not a reference, i.e., if this intermediate link is not present. In that case the identity declaration establishes the identifier as a constant, which can be changed only by redeclaring it. An example of a declaration that establishes π as a constant 3.14 is **real $\pi = 3.14$** , which gives rise to the following scheme:

identifier $\pi \rightarrow 3.14$.

This careful distinction permits, in particular, the definition of constant and variable procedures. For example, the declaration **proc $f = (\text{real } x, \text{real } y)$ real: $(x + y)/2 - \text{sqrt}(x \times y)$** establishes f as a constant, as opposed to **proc $f: = (\text{real } x, \text{real } y)$ real: $(x + y)/2 - \text{sqrt}(x \times y)$** , which defines a variable procedure. In the latter case we can, at another point in the program, assign some other value of mode **proc (real, real) real to f** ; for example, we can write **$f: = (\text{real } x, \text{real } y)$ real: $(x + y)/2$** .

A number of standard statements are available in Algol 68: assignment, e.g., $x := (a + b)/2$; repetitive, e.g., **for i from 2 to n do $f := f \times i$; go to, e.g., go to loop**; conditional, e.g., **if $x \geq y$ then go to label else go to end fi**, etc. In addition to the conventional serial statement execution, it is possible to specify parallel or *collateral* execution. In the latter case, execution of statements is merged in time in a way to be specified by the implementation. Parallel programming facilities in Algol 68 include elementary means of control or synchronization of collateral execution. These are language-defined values called "semaphores."

The Algol 60 concept of a block appears in a more general form in Algol 68 as a *range*. An example of a range is a sequence of declarations and statements placed between generalized parentheses. Examples of pairs of these parentheses are **begin** and **end**, **if** and **then**, **then** and **else**, **else** and **fi**, etc. References possessed by the identifiers declared in a range may be local to that range. Since the hardware representation of a reference is a memory location, storage is allocated dynamically to local variables; i.e., storage for local variables of a range is deallocated when leaving that range. In addition to these stack-controlled values, Algol 68 also has

values whose lifetime does not fit into the last-in-first-out principle of a stack. Values of this sort are stored in a randomly organized memory region called the heap.

REFERENCES

1969. van Wijngaarden, A., et al. "Report on the Algorithmic Language ALGOL 68." Springer-Verlag *Numerische Mathematik*, 14.
1971. Lindsey, C. H., and S. G. van der Meulen. *Informal Introduction to ALGOL 68*. Amsterdam: North-Holland.

S. ALAGIĆ

ALGORITHM

For articles on related subjects see **ALGORITHMS, ANALYSIS OF; ALGORITHMS, THEORY OF; ERRORS; ERROR ANALYSIS; MARKOV ALGORITHM; PARALLEL ALGORITHM; PROGRAM CORRECTNESS, PROOF OF; SCHEDULING ALGORITHM; and TURING MACHINE.**

In discussing problem solving, we presuppose both a problem and a device to be used in solving the problem. The problem may be mathematical or nonmathematical in nature, simple or complex. The basic requirements for a well-posed problem are that (1) the known information is clearly specified; (2) the solution is specified to the extent that it can be determined when the problem has been solved; and (3) the problem does not change during its attempted solution. The second requirement does not mean that the solution to the problem is known a priori, but only that we can determine when we have reached the solution. For example, in some numerical problems we obtain repeated approximations to the answer, terminating the solution process when two successive approximations are "sufficiently close" together. We can specify in the problem statement the exact meaning of "sufficiently close," without knowing the exact answer. The device to be used for problem solution may be man or machine, or a combination of the two.

Definition. Given both the problem and the device, an *algorithm* is the precise characterization of a method of solving the problem, presented in a

ALGORITHM

language comprehensible to the device. In particular, an algorithm is characterized by these properties:

1. Application of the algorithm to a particular input set or problem description results in a finite sequence of actions.
2. The sequence of actions has a unique initial action.
3. Each action in the sequence has a unique successor.
4. The sequence terminates with either a solution to the problem, or a statement that the problem is insoluble.

We illustrate these concepts with an example: "Find the square root of the real number x ." As it is stated, this problem is algorithmically either trivial or unsolvable, owing to the irrationality of most square roots. If we accept " $\sqrt{2}$ " as the square root of 2, for example, the solution is trivial: The answer is the square root sign ($\sqrt{}$) concatenated with the input. In SNOBOL, the entire algorithm is

```
OUTPUT = '√' INPUT
END
```

However, if we want a decimal expression, then the square root of 2 can never be exactly calculated. Hence, the requirement of a finite number of actions is violated.

A modified statement of the problem is more suited to our purposes. "Find the positive square root, to four decimal places, of the real number x ." This statement has three useful properties:

1. It explicitly names the *positive* square root as the desired one, whereas the earlier statement left that quality ambiguous.
2. It eliminates the string " \sqrt{x} " as a problem solution.
3. By stating "four decimal places" (or any other fixed number of places), it provides a test for termination.

A possible method of solution is

- (a) Choose a number y and compute y^2 .
- (b) If $|y^2 - x| < 5 \times 10^{-5}$, the solution is y ; if not, return to step (a).

This method fails to be an algorithm, since no procedure is specified for choosing either the initial value y or subsequent values. Moreover, even if there is a solution, there is no guarantee

that this method will find it.

Now consider another method:

1. Let $y = 1$.
2. Compute y^2 .
3. If $|y^2 - x| < 5 \times 10^{-5}$, the solution is y , **HALT**; if not, go to step 4.
4. Replace y by $((x/y) + y)/2$; go to step 2. (This procedure is a special case of a general technique known as the Newton-Raphson technique.)

This method has the precise definition of each step required of an algorithm. Moreover, whenever applied to a nonnegative real number x , the method will produce the proper solution in a finite number of steps. However, whenever applied to a negative number, the method will endlessly recompute y without recognizing the futility of the task. This is typical of a class of methods called *semi-algorithms*: They will halt in a finite number of steps if the problem posed has a solution, but will not necessarily halt if there is no solution.

To **transform** the given method into an algorithm, two things must be done:

- (a) Add a step, (0); if $x < 0$, there is no solution; **HALT**; and
- (b) Rewrite the given method in a language suitable for the proposed device. (For English-speaking people, the given language is satisfactory; for a computer, a programming language **must** be used. For example, the following algorithm in Basic is suitable for computers utilizing that language, with the data set (3, 107, 1, 0, -4).)

```
10 READ X
20 IF X<0 THEN 80
30 LET Y = 1
40 LET Z = Y^2
50 IF ABS (X-Z) < 0.00005 THEN 100
60 Y = ((X/Y) + Y)/2
70 GO TO 40
80 PRINT "THERE IS NO SOLUTION FOR " X "."
90 GO TO 10
100 PRINT "THE SQUARE ROOT OF " X " IS " Y "."
105 GO TO 10
110 DATA 3
111 DATA 107
112 DATA 1
113 DATA 0
114 DATA -4
120 END
```

Note that statements 110 through 114 specify the data set to be used with the algorithm and are not part of the algorithm itself. If the algorithm is applied to this particular data set, the result will be

```
THE SQUARE ROOT OF 3      IS 1.73205 .
THE SQUARE ROOT OF 107    IS 10.3441 ,
THE SQUARE ROOT OF 1      IS 1 .
THE SQUARE ROOT OF 0      IS 3.90625E-3 .
THERE IS NO SOLUTION FOR -4 .
```

Significance of Algorithms. While the concept of an algorithm is useful in crystallizing the informal notation of a “method of solution” for a problem, it has a much deeper significance. Whereas it was at one time assumed that any properly stated mathematical problem was solvable, mathematicians in the 1920s began to question this, asking what precisely it meant to say that we could “solve a problem” or “compute a function.” Several important areas of mathematics have resulted from attempts to answer these questions, including the theory of Turing machines and the theory of algorithms. All the concepts proposed proved to be equivalent: Any problem that is solvable according to one concept is solvable according to all other concepts. Thus, while the algorithm, properly formalized, may not be the only way to solve problems, it appears to be essentially the only way that the human intellect in its present stage of development can comprehend.

Quality Judgments on Algorithms. Any computer program is at least a semi-algorithm, and any program that always halts is an algorithm. (Of course it may not solve the problem for which the programmer intended it.) Given a solvable problem, there are many algorithms (programs) to solve it, not all of equal quality. The primary practical criteria by which the quality of an algorithm is judged are time and memory requirements, accuracy of solution, and generality. To cite an extreme example, since a properly defined game of chess has only a finite number of possible moves, there exists an algorithm to determine the “perfect” chess game. Simply examine all possible move sequences, in some specified order. Unfortunately, the time required to execute any algorithm based on this idea is measured in billions of years, even at today’s computer speeds. The memory requirements for such an algorithm are similarly overbearing.

On a more practical plane, several numerical methods for solving problems fail to yield satisfactory algorithms because the rate of convergence is

so slow that thousands or millions of iterations may be needed to determine the answer. For other numerical methods, rounding or truncation errors may accumulate so rapidly that they destroy the answer.

There is often a trade-off in time and memory requirements, which must be settled pragmatically. The simplest case of this arises in the computation and repeated use of a complicated function. If the computation of each function value is sufficiently complex, then in repeated usage much time may be saved by **precomputing** a table of values and using table lookup techniques. However, such a table may require sufficient additional memory space that this becomes a critical factor. Thus, one may have to sacrifice some speed to stay within available memory bounds.

The accuracy of an algorithm is a characteristic often more closely related to time than to memory requirements. For example, the square root algorithm previously presented is not very accurate. (It yields 0.00390625 as the square root of zero.) Its accuracy may be improved by changing the test constant in line 50, at the cost of a longer run time. Further improvement may be obtained from the corresponding algorithm in double-precision Fortran, at a cost of both run time and additional memory space. In each case the basic algorithmic concept is unchanged.

Altering the basic algorithmic concept may provide an improved algorithm to accomplish a given task. For example, three multiplications and two additions are required to evaluate the quadratic expression $ax^2 + bx + c$ in the order $((ax^2) + (bx)) + c$. Changing the concept of the evaluation algorithm to $((ax) + b)x + c$ eliminates one multiplication, resulting in a more efficient process. This will improve the speed of solution of the problem, and probably also improve the accuracy of the result.

The remaining important characteristic of an algorithm is its generality. While there are occasions when an algorithm is needed to solve a single isolated problem, more often algorithms are designed to handle a range of input data. Generality, like accuracy, is often attained at the cost of speed and memory requirements. A general polynomial root finder is more **costly** in both time and storage than an algorithm for extracting the roots of a quadratic equation. But the increased generality may justify the cost. This is a pragmatic decision. In another example, an information retrieval system based on a free vocabulary is generally more expensive to design and operate than one based on a fixed

ALGORITHM, MARKOV

or coded vocabulary. But the difference in utility may far outweigh the additional cost burden.

R. R. KORFHAGE

ALGORITHM, MARKOV. *See* MARKOV ALGORITHM.

ALGORITHM, PARALLEL. *See* PARALLEL ALGORITHM.

ALGORITHM, SCHEDULING. *See* SCHEDULING ALGORITHM.

ALGORITHMS, ANALYSIS OF

For articles on related subjects see ALGORITHM; ALGORITHMS, THEORY OF; COMPUTATIONAL COMPLEXITY; ITERATION; and RECURSION.

For article on related term see GRAPH THEORY.

The analysis of algorithms can be partitioned into two areas: algorithm complexity and problem complexity. The former is concerned with consideration of a specific algorithm for a problem and the analysis of its behavior with respect to the amount of memory space, time, or other resource used. The latter is concerned with the class of all algorithms for a particular problem and the determination of the minimum requirements of the problem with respect to space and time or other resources. Such analyses are second in importance only to the determination of the correctness of an algorithm. They provide the means to choose intelligently and improve algorithms.

Contrary to one's intuition, the advent of the electronic computer has made the efficiency of algorithms a topic of utmost concern. One might suspect that as the speed of computers increases, the effects of the efficiency of the algorithms decrease. Actually, just the opposite is true. The reason for this is that the asymptotic behavior of the algorithm becomes more important, as we will now illustrate.

With a problem we associate an integer, which we call the size of the problem. For example, the size

of a matrix multiplication problem is the dimension of the matrix, the size of a graph problem is the number of edges, etc. The growth rate of the execution time of the algorithm is determined as a function of the size of the problem. The limiting behavior of the growth rate is called the asymptotic growth rate. For example, the asymptotic behavior of the function $17 + 5n + 2n^2$ is $2n^2$, since, for sufficiently large n , $2n^2$ approximates $17 + 5n + 2n^2$ to arbitrary accuracy. For $n = 100$, the lower-order terms account for less than 3%.

In performing a hand computation, the size of the problem is small, and consequently the asymptotic growth rate is unimportant. On such small problems most algorithms perform reasonably well. However, on a high-speed computer, the problem size normally encountered is large and the asymptotic growth rate becomes important. Given two algorithms with growth rates n^2 and 2^n , for problems up to size 6, the difference in execution times is never more than a factor of 2. However, with a computer, a problem of size 100 might be encountered. In this case, the n^2 algorithm is easily executed, whereas the 2^n algorithm would require centuries to compute. This example illustrates why in the past ten years a tremendous effort has been devoted to analysis of algorithms.

Algorithm Complexity. Space and time are the most important considerations of algorithm complexity. Since both are limited, it is advisable to determine how much space and time an algorithm requires. An algorithm that requires relatively little memory space for execution may have a greater running time than another algorithm that requires more space, while both algorithms may provide a solution to the same problem. Thus, there is frequently a trade-off between space and time.

As an example of a space-time trade-off, consider an algorithm that requires the storage of an undirected graph. (An undirected graph is a set V of n vertices, $V = \{v_1, v_2, \dots, v_n\}$, and a set E of edges, where an edge is an unordered pair of vertices.) The algorithm stores the graph as an $n \times n$ connection matrix A , where

$$a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \text{ is an edge in } E; \\ 0 & \text{otherwise.} \end{cases}$$

This requires n^2 words of memory, regardless of the number of edges. Assume that the algorithm is used only for planar graphs. (A planar graph is an

undirected graph that can be drawn on a plane surface so that no edges intersect.)

Let G be a planar graph with p vertices. Then G can be represented in the computer by a linked list of n vertices where the data structure for each vertex v_i is a linked list of all vertices adjacent to v_i . Since each edge (v_i, v_j) of G is stored twice (v_j is on the list of vertices adjacent to v_i , and v_i is on the list of vertices adjacent to v_j), the memory required to store the list representation of G is proportional to the number of edges. For planar graphs it can be shown that the number of edges is bounded by $3n - 6$, where n is the number of vertices. Thus, the memory required is bounded by $C \times n$, where C is a constant, rather than the n^2 that was required for the connection matrix representation. If the algorithm is required to determine if vertex v_i is connected to vertex v_j , then a trade-off between space and time occurs, since only one operation is needed with the connection matrix representation, whereas the list representation requires searching the entire list of vertices adjacent to v_i to see if v_j is on the list.

A frequency analysis of an algorithm reveals the number of times certain parts of the algorithm are executed. Such an analysis indicates which parts of the algorithm consume large quantities of time and hence where efforts should be directed toward improving the algorithm. For example, the following section of Fortran-like code calculates

$$\sum_{i=1}^{N+1} a_i x^{i-1}$$

and stores the result in T .

```

      DIMENSION S(N), A(N + 1)
      DO 10 I = 1,N
1     Y = 1.0                               N
      DO 20 J = 1,I
2     Y = Y * X                             N(N + 1)/2
20    CONTINUE
3     S(I) = A(I + 1)*Y                     N
10    CONTINUE
4     T = A ( 1 )                           1
      DO 30 I = 1,N
5     T = T + S(I)                           N
30    CONTINUE

```

The program is poorly written and just about every statement can be changed to decrease the amount of time required. To the right of each assignment statement is the number of times it is executed. As N increases, the program spends proportionately more and more time executing statement 2 than it does for statements 1, 3, 4, or 5. Thus, it is really futile to try

to improve the program by decreasing the time spent executing the latter statements without first decreasing the time spent executing statement 2. The portion of the program containing statement 2 can be improved by using Horner's rule for polynomial evaluation. Again, the number of times each assignment statement is executed is given at its right.

```

      DIMENSION A(N + 1)
1     T = A(N + 1)                               1
      DO 10 I = 1,N
2     T = T*X + A(N + 1 - I)                     N
10    CONTINUE

```

To determine the actual execution time of an algorithm in seconds requires a knowledge of the operation times for each instruction of the computer on which the algorithm is to be executed and how the compiler generates code. In order to avoid becoming involved in the specific details of operation of a particular computer, it is customary to find upper and lower bounds c_1 and c_2 such that the execution time of every instruction is between c_1 and c_2 . Then the execution time of an algorithm can be estimated from a count of the number of operations that are executed. This frees the analysis of the algorithm from peculiarities of individual computers.

Frequently the time required by an algorithm is data dependent. In this case one of two types of analyses is possible. The first is called the "worst-case analysis," in which that set of data of given size requiring the most work is determined and the behavior of the algorithm is analyzed for that specific set of data. The other alternative is to assume a probability distribution for the possible input data and compute the distribution of the execution time as a function of the input distribution. Usually, this computation is so difficult that only the expected or average execution time as a function of size is computed. This is called the "average case analysis."

Problem Complexity. In problem complexity we are concerned with analyzing a problem rather than an algorithm. The analysis provides us with lower bounds on the amount of time and space required for a solution to the problem, independent of the algorithm used. The lower bounds may be either "worst case" or "average case" bounds. These lower bounds can serve as an indication of how well an algorithm fits the problem and whether it can be improved. For example, such an analysis shows that any algorithm that evaluates an arbitrary n -degree

ALGORITHMS, ANALYSIS OF

polynomial represented by its coefficients requires at least n multiplications and n additions. Thus, Homer's rule (given above) cannot be improved upon.

On the other hand, an analysis of matrix multiplication gives a lower bound of order n^2 operations for multiplying two matrices of dimension n . The usual matrix multiplication algorithm has an asymptotic growth rate of order n^3 . Thus, there is substantial interest in trying either to find a better lower bound or to improve on the current matrix multiplication algorithms. At the current state of knowledge the fastest algorithm has an asymptotic growth rate of order $n^{2.81}$, and thus there is a large gap between the best-known lower bound and the performance of the best-known algorithm.

In problem analysis, it is often important to consider the frequency of occurrence of a specific operation. The reason for this is that reducing the number of occurrences of a specific operation can lead to a recursive algorithm with a lower asymptotic growth rate. Consider multiplying two n -digit numbers, where n is a power of 2. The usual algorithm learned in elementary school requires on the order of n^2 operations. A recursive method of multiplying two n -digit numbers x and y is to write $x = a10^{n/2} + b$ and $y = c10^{n/2} + d$, where a , b , c , and d are $n/2$ -digit numbers. Compute ab , cd , and $ad + bc$. Then

$$xy = ab10^n + (ad + bc)10^{n/2} + cd.$$

The problem of computing xy is reduced to the problem of computing ab , cd , and $ad + bc$, which are computed by the three multiplications ab , cd , and $(a + c)(b + d)$. The formula $ad + bc$ is obtained by $(a + c)(b + d) - ad - cd$. Let $T(n)$ be the time to compute the product of two n -digit numbers. Then $T(n) \simeq 3T(n/2) + kn$, where the $3T(n/2)$ is the time to compute the three multiplications, k is a nonnegative constant, and kn is the time to compute the necessary sums. Successively applying the formula above to each product, we obtain

$$\begin{aligned} T(n) &\simeq kn(1 + (3/2) + (3/2)^2 + \dots + (3/2)^{\log_2 n}) \\ &\simeq 3kn^{\log_2 3} \simeq 3kn^{1.58}. \end{aligned}$$

The asymptotic growth rate is of order $n^{1.58}$ rather than the n^2 of the more elementary method. The important observation is that in computing ab , cd , and $ad + bc$, the number of multiplications was reduced from four to three at the expense of increasing the number of additions from one to four.

The reason for doing this is that the exponent in the asymptotic growth rate is affected by the number of multiplications, whereas the number of additions affects only the constant.

A major difficulty with problem analysis is that it is concerned with the class of all algorithms for a given problem. One no longer can postulate a computer with a given structure and operation set. Instead, one must envision an abstract computer that is sufficiently general to encompass any physically implementable algorithm. The difficulties involved are of such magnitude that one is forced to obtain bounds for certain limited classes of programs. For example, sorting n integers can be shown to require $n \log n$ operations if restricted to the class of algorithm that sorts by binary comparisons. This follows from the simple information theoretic argument that there are $n!$ possible permutations of n items, and each comparison can at best divide the set of possible permutations by a factor of 2. Since the asymptotic growth rate of $\log(n!)$ is $n \log n$, it takes at least $n \log n$ comparisons to determine uniquely the actual permutation. Of course, if one sorts by some method other than by comparisons (radix sort, for example), then the bound is no longer valid.

A typical assumption for a class of programs might be that the computation uses only the arithmetic operations of addition, subtraction, multiplication, and division. When this is done, it is necessary to specify the underlying algebraic structure. For example, the complexity of computing an algebraic expression may depend on whether the underlying structure is the rational, real, or complex number system.

One of the most powerful techniques for establishing results of this nature is due to Winograd, who showed that any algorithm for computing the product of an arbitrary vector X times a matrix A requires a number of multiplications at least as great as the number of nondependent columns of A . It immediately follows from this result that Horner's rule evaluates arbitrary n -degree polynomials with the minimum number of multiplications. Let $X = (x^n, x^{n-1}, \dots, x, 1)$ and let $A = (a_{n+1}, a_n, \dots, a_1)^T$. Then

$$X \cdot A = \sum_{i=1}^{n+1} a_i x^{i-1}.$$

X has n nondependent columns, which implies that n multiplications are required. The result requires that the algorithm evaluate any polynomial, given its coefficients. Specific polynomials can often be evaluated with fewer multiplications. Similarly, if the

polynomial is specified by parameters other than its coefficients, a saving in the number of multiplications is possible.

The one facet of problem complexity that is probably the most intriguing is the lack of nontrivial lower bounds for various problems. Almost all known lower bounds are either linear in the size of the problem or have been obtained by restricting the classes of algorithms. The notable exceptions are lower bounds obtained by the diagonalization techniques of recursive function theory. One of the major goals of computer scientists working in the analysis of algorithms is to close the gap in our knowledge of problem complexity. Hopefully, the next decade will provide powerful new tools in the area or startling improvements in the efficiency of algorithms.

REFERENCE

1968, 1969, 1973, Knuth, D. E. *The Art Of Computer Programming*, vols. 1, 2, 3. Reading, Mass.: Addison-Wesley.

J. E. HOPCROFT AND J. E. MUSINSKI

ALGORITHMS, THEORY OF

For articles on related subjects see **ALGORITHM**; **ALGORITHMS, ANALYSIS OF**; **COMPUTABILITY**; **COMPUTATIONAL COMPLEXITY**; **DECIDABILITY**; **FORMAL LANGUAGES**; and **TURING MACHINE**.

For article on related term see **PROCEDURE**.

The meaning of the word "algorithm," like the meaning of most other words commonly used in the English language, is somewhat vague. In order to have a *theory of algorithms*, we need a mathematically precise definition of an algorithm. However, in giving such a precise definition, we run the risk of not reflecting exactly the intuitive notion behind the word. The finding of a mathematically precise replacement of the notion of algorithm was the earliest problem in the theory of algorithms. Many authors have tried to capture the essence of the intuitive notion of an algorithm. We give four examples.

Hermes (1965). "An algorithm is a general procedure such that for any appropriate question the answer can be obtained by the use of a simple computation according to a specified method. . . . [A]

general procedure [is] a process, the execution of which is clearly specified to the smallest details."

Minsky (1967). "... an effective procedure is a set of rules which tell us, from moment to moment, precisely how to behave."

Rogers (1967). "... an algorithm is a clerical (i.e., deterministic, bookkeeping) procedure which can be applied to any of a certain class of symbolic inputs and which will eventually yield, for each such input, a corresponding symbolic output."

Hopcroft and Ullman (1969). "A procedure is a finite sequence of instructions that can be mechanically carried out, such as a computer program. . . . A procedure which always terminates is called an *algorithm*."

Note that what *Hermes* calls a "general procedure" is what *Minsky* calls an "effective procedure" is what *Hopcroft and Ullman* call a "procedure." Other terms are also used in the literature, and some authors use the word "algorithm" to denote any procedure whatsoever. In the remainder of this article the *Hopcroft and Ullman* terminology will be used.

An important fact to note is that the notion of a procedure cannot be divorced from the environment in which it operates. What may be a procedure in certain situations, may not be considered a procedure in other situations. For example, the instructions of a computer program are not usually understood by most people. Alternatively, the description of a chess game that appears in a newspaper is a perfectly clear algorithm for a chess player who wants to reproduce the game, but it is quite meaningless to people who do not play chess. Thus, when we talk about a procedure as a finite sequence of instructions, we assume that whoever is supposed to carry out those instructions, be it man or machine, understands them in the same way as whoever gave those instructions.

Another sense in which the environment influences the notions of procedure and algorithm is indicated by the following examples. If the instruction requires us to take the integral part of the square root of a number, such an instruction can be carried out if we are dealing with positive integers only, but it cannot always be carried out if we are dealing with both positive and negative integers. Thus, the same set of instructions may or may not be a procedure, depending on the subset of integers for which it is intended. Alternatively, we can easily give a procedure that, given an integer x , keeps subtracting 1 until 0 is reached and then stops. Such a procedure will be an algorithm if we intend to use it for positive

ALGORITHMS, THEORY OF

integers only, but it will not be an algorithm if we also intend to apply it to negative integers.

The recognition of whether or not a sequence of instructions is a procedure or an algorithm is a subjective affair. No precise theory can be built on the vague definitions given above. In trying to build a precise theory, one must examine the situations in which the notion of algorithm is used. In the theory of computation, one is mainly concerned with algorithms that are used either for computing functions or for deciding predicates.

A function f with domain D and range R is a definite correspondence by which there is associated with each element x of the domain D (referred to as the "argument") a single element $f(x)$ of the range R (called the "value"). The function f is said to be "computable" (in the intuitive sense) if there exists an algorithm which, for any given x in D , provides us with the value $f(x)$.

A *predicate* P with domain D is a property of the elements of D , which each particular element of D either has or does not have. If x in D has the property P , we say that $P(x)$ is true; otherwise, we say that $P(x)$ is false. The predicate P is said to be *decidable* (in the intuitive sense) if there exists an algorithm which, for any given x in D , provides us with a definite answer to the question of whether or not $P(x)$ is true.

The computability of functions and the decidability of predicates are very closely related notions because we can associate with each predicate P a function f with range $\{0,1\}$ such that, for all x in the common domain D of P and f , $f(x) = 0$ if $P(x)$ is true and $f(x) = 1$ if $P(x)$ is false. Clearly, P is decidable if and only if f is computable. For this reason we will hereafter restrict our attention to the computability of functions.

A further restriction customary in the theory of algorithms is to consider only functions whose domain and range are both the set of nonnegative integers. This is reasonable, since in those situations where the notion of a procedure makes any sense at all, it is usually possible to represent elements of the domain and the range by nonnegative integers. For example, if the domain comprises pairs of nonnegative integers, as in the case with an arithmetic function of two arguments, we can represent the pair (a, b) , by the number $2^a 3^b$ in an effective one-to-one fashion. If the domain comprises strings of symbols over an alphabet of 15 letters, we can consider the letters to be **nonzero** hexadecimal digits, and assign that nonnegative integer to a string that is denoted by the string in the hexadecimal notation. The device of representing elements of a set D by nonnegative

integers is referred to as "arithmetization" or "Gödel numbering," after the logician K. Gödel, who used it to prove the undecidability of certain predicates about formal logic. From now on we will be exclusively concerned with functions whose domain and range are subsets of the set of nonnegative integers.

In order to show that a certain function is computable, it is sufficient to give an algorithm that computes it. But without a precise definition of an algorithm, all such demonstrations are open to question. The situation is even more uncertain if we want to show that a given function is **uncomputable**, i.e., that no algorithm whatsoever computes it. In order to avoid such uncertainty, we need a mathematically precise definition of a computable function,

It is clear from the way in which algorithms are discussed above that for a proper algorithm we ought to be able to construct a machine that carries out the instructions of the algorithm. One possible way of making precise the concept of a computable function is to define an appropriate type of machine, and then define a function to be computable if and only if it can be computed by such a machine. This has indeed been done. The machine usually used for this purpose is the so-called Turing machine. This simple device has a tape and a read-write head, together with a control that may be in one of finitely many states. The tape is used to represent numbers. A function f is called computable if there exists a Turing machine that, given a tape representing an argument x , eventually halts with the tape representing the value $f(x)$. Since a precise definition of a Turing machine can be given, the notion of a computable function has become a precise mathematical notion.

The question arises whether or not it is indeed the case that a function is computable in the intuitive sense if and only if it is computable by a Turing machine. The claim that this is true is usually referred to as "Church's thesis" (sometimes as Turing's thesis). Such a claim can never be "proved," since one of the two notions whose equivalence is claimed is mathematically imprecise. However, there are many convincing arguments in support of Church's thesis, and an overwhelming majority of workers in the theory of algorithms accept its validity. One of the strongest arguments in support of Church's thesis is the fact that all of the many diverse attempts at precisely defining the concept of computable function have ended up with defining exactly the same set of functions.

Given a precise definition of a computable function, it is now possible to show for particular functions that they are computable. Conversely, it becomes possible to prove that certain functions are not computable. We will give two examples.

Example 1. Consider the following problem. Give an algorithm that, for any Turing machine, decides whether or not the machine eventually stops if it is started on an empty tape. This problem is called the "blank-tape halting problem." The required algorithm would be considered a *solution* of the problem. A proof that there is no such algorithm would be said to show the (effective) *unsolvability* of the problem.

The blank-tape halting problem is in fact unsolvable. This is proved by rephrasing the problem into a problem about the computability of a function, as follows: Turing machines can be Gödel-numbered in an effective manner; i.e., there exists an algorithm which for any Turing machine will give its Gödel number. Furthermore, this can be done in such a way that every nonnegative integer is the Gödel number of some Turing machine. Let f be the function defined as follows:

$$f(x) = \begin{cases} 0 & \text{if } n \text{ is the Gödel number of a Turing} \\ & \text{machine that eventually stops if} \\ & \text{started on the blank tape;} \\ 1 & \text{otherwise.} \end{cases}$$

It is easy to see that f is computable if and only if the blank-tape halting problem is solvable. The unsolvability of the blank-tape halting problem is proved by showing that the assumption that f is computable leads to a contradiction.

Example 2. Our second example indicates that there are unsolvable problems in classical mathematics. The following problem is known as "Hilbert's tenth problem" (after the German mathematician David Hilbert, 1862-1943):

Given a diophantine equation [an equation of the form $E = 0$, where E is a polynomial with integer coefficients; e.g., $xy^2 - 2x^2 + 3 = 0$] with any variables, give a procedure with which it is possible to decide after a finite number of operations whether or not the equation has a solution in integers,

Although this problem was stated by Hilbert in 1900 (long before there was such a thing as a theory of algorithms), it has only been recently that the

Russian mathematician I. Matijasevitch has shown it to be unsolvable.

That there are clearly defined problems, like the two given above, that cannot be solved by any computer-like device is probably the most striking aspect of the theory of algorithms. A whole superstructure has been built on such results, and there are methods to find out not only whether something is **uncomputable**, but also how badly it is **uncomputable** (see Rogers, 1967).

A typical question that one may ask is the following: Suppose we had a device which, for any given Turing machine, told us whether or not the Turing machine will eventually stop on the blank tape. Can we write an "algorithm" that makes use of this device and solves Hilbert's tenth problem? It has been known for some time that such an "algorithm" exists. In this sense, Hilbert's tenth problem is *reducible* to the blank-tape halting problem. It is the proof that the reverse is also true which gave us the unsolvability of Hilbert's tenth problem. Two problems that are both reducible to the other are said to be "equivalent." Most of the theory of algorithms has, until recently, concerned itself with questions of the reducibility and equivalence of various unsolvable problems.

In recent years a new trend has developed. Much of the activity in the theory of algorithms began to concern itself with computable functions, decidable predicates, and solvable problems. Questions about the nature of the algorithms, the type of devices that can be used for the computation, and about the difficulty or complexity of the computation have been investigated and are discussed in other articles.

REFERENCES

- 1965. Hermes, H. *Enumerability, Decidability, Computability*. Berlin, Germany: Springer Verlag.
- 1967. Minsky, M. *Computation: Finite and Infinite Machines*. Englewood Cliffs, N.J.: Prentice-Hall.
- 1967. Rogers, H. *Theory of Recursive Functions and Effective Computability*. New York: McGraw-Hill.
- 1969. Hopcroft, J. E., and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Reading, Mass. : Addison-Wesley.

G. T. HERMAN

ALLOCATION, STORAGE

ALLOCATION, STORAGE. See **STORAGE ALLOCATION.**

AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES (AFIPS)

For articles on related terms see **AMERICAN SOCIETY FOR INFORMATION SCIENCE; ASSOCIATION FOR COMPUTING MACHINERY; ASSOCIATION FOR EDUCATIONAL DATA SYSTEMS; INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING; INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS-COMPUTER SOCIETY; and SOCIETY FOR INDUSTRIAL AND APPLIED MATHEMATICS.**

Purpose. The American Federation of Information Processing Societies is a national federation of professional societies established to represent the member societies on an international level and for the advancement and diffusion of knowledge of the information processing sciences. Toward the latter end it engages in appropriate literary and scientific activities. High on the list of its original goals was the provision of a complete, responsible, and effective public information program for the information processing community, and AFIPS has performed this function primarily and most effectively through sponsorship of the Spring and Fall Joint Computer Conferences, which as of June 1973 were merged into an annual National Computer Conference. AFIPS represents the United States in a variety of international information processing activities, including IFIP (International Federation for Information Processing). AFIPS acts as national spokesman for the information processing community in matters dealing with or affected by computing, data processing, and related sciences.

How Established. AFIPS was organized as an unincorporated society on May 10, 1961. It was the outgrowth of the National Joint Computer Committee, which had been established ten years earlier to sponsor the Joint Computer Conferences. The AFIPS founding societies were the American Institute of Electrical Engineers, Institute of Radio Engineers (which later merged into the Institute of Electrical and Electronic Engineers), and the Association for Computing Machinery.

The presidents who have held office in the National Joint Computer Committee and AFIPS are

Morton M. Astrahan, 1956-1958
Harry H. Goode, 1959-1960
Willis Ware, 1961-1962
J. D. Madden, 1963
Edwin L. Harder, 1964-1965
Bruce Gilchrist, 1966-1967
Paul Armer, 1968
Richard I. Tanaka, 1969-1970
Keith W. Uncapher, 1971
Walter L. Anderson, 1972
George Glaser, 1973-1975
Anthony Ralston, 1975-

Organizational Structure. There are two classes of AFIPS participation: member societies, which have a principal interest in computers and information processing, and affiliated societies, which, although not principally concerned with computers and information processing, do have a major interest in this field. A minimum membership of 1,500 is required for admission to either class of membership.

In 1975 the 15 constituent societies of AFIPS were

The Association for Computing Machinery, Inc. (ACM)
The Institute of Electrical & Electronics Engineers, Inc. (IEEE)
Data Processing Management Association (DPMA)
Society for Computer Simulation (SCS)—formerly Simulation Councils, Inc. (SCI)
The American Society for Information Science (ASIS)
Association for Computational Linguistics (ACL)
Society for Information Display (SID)
Special Libraries Association (SLA)
The American Institute of Certified Public Accountants (AICPA)
American Statistical Association (ASA)
Society for Industrial and Applied Mathematics (SIAM)
American Institute of Aeronautics and Astronautics (AIAA)
Instrument Society of America (ISA)
Association for Educational Data Systems (AEDS)
Institute of Internal Auditors (IIA)

AMERICAN SOCIETY FOR INFORMATION SCIENCE (ASIS)

The Federation is managed by its Board of Directors. Each member society has one to three directors, depending on size; each affiliated society has one director. The President is the principal officer of the Federation and the Executive Director is the senior paid officer. Other AFIPS officers include a vice-president, secretary, and treasurer. Meetings of the Board of Directors are held at least once a year to elect member and associate member societies, to act on constitutional amendments, and to conduct other pertinent business.

The headquarters of AFIPS is located at 210 Summit Avenue, Montvale, New Jersey 07645.

Technical Program. The chief contribution of AFIPS to the professional community has been made through its sponsorship each year of both Spring and Fall Joint Computer Conferences, which offered technical sessions in the widest possible spectrum of subjects, and an accompanying exhibit of the latest equipment and literature relevant to the information sciences. In 1973 these conferences were replaced by an annual National Computer Conference, which continues to provide both the professional community and the interested public with the latest information regarding developments in the field of information technology. AFIPS also sponsors small meetings and workshops on specialized topics. Several of these have resulted in publications.

The Harry Goode Memorial Award was authorized in 1964, and since that time has been presented annually to an individual in recognition of outstanding achievement in the field of information processing. The recipients of this award are

Howard Hathaway Aiken, 1964
George Robert Stibitz and Konrad Zuse, 1965
J. Presper Eckert and John William Mauchly, 1966
Samuel Nathan Alexander, 1967
Maurice Vincent Wilkes, 1968
Alston Scott Householder, 1969
Grace Murray Hopper, 1970
Allen Newell, 1971
Seymour R. Cray, 1972
(No award, 1973)
Edsger W. Dijkstra, 1974.

The AFTPS Press publishes proceedings of annual conferences and other publications of interest to its members and to the lay public, and is also the distributor of IFIP publications in the United States.

I. L. AUERBACH

AMERICAN SOCIETY FOR INFORMATION SCIENCE (ASIS)

For article on related subject see **AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES**.

Purpose. The American Society for Information Science is a not-for-profit professional association organized for scientific, literary, and educational purposes, and dedicated to the creation, organization, dissemination, and application of knowledge concerning information and its transfer, with particular emphasis on the applications of modern technologies in these areas.

An auxiliary purpose of the Society is to provide its members with a variety of channels of communication within and outside the profession, including meetings and publications, and with a service organization to help them in their professional development and advancement.

How Established. ASIS was founded on March 13, 1937, as the American Documentation Institute (ADI) when Watson Davis, director of Science Service (which was operated out of the National Academy) and one of the first Americans to become interested in documentation as a separate field of endeavor, invited approximately 35 documentalists colleagues to meet with him at the National Academy of Sciences. ADI was made up of individuals nominated by and representing affiliated scientific and professional societies, foundations, and government agencies, of which there were 68 in 1937. In 1952, the bylaws were amended to admit individual as well as institutional members. By vote of the membership on Jan. 1, 1968, the name was changed to American Society for Information Science, to indicate its concern with all aspects of the information-transfer process.

The following individuals have held the office of president:

Watson Davis, 1937–1943
Keyes D. Metcalf, 1944
Waldo G. Leland, 1945
Watson Davis, 1946
Waldo G. Leland, 1947
Vernon D. Tate, 1948–1949
Luther H. Evans, 1950–1952
E. Eugene Miller, 1953
Milton O. Lee, 1954
Scott Adams, 1955

AMERICAN SOCIETY FOR INFORMATION SCIENCE (ASIS)

Joseph Hilsenrath, 1956
James W. Perry, 1957
Herman H. Henkle, 1958
Karl F. Heumann, 1959
Cloyd Dake Gull, 1960
Gerald J. Sophar, 1961
Claire K. Schultz, 1962
Robert M. Hayes, 1963
Hans Peter Luhn, 1964
Laurence B. Heilprin, 1964-1965
Harold Borko, 1966
Bernard M. Fry, 1967
Robert S. Taylor, 1968
Joseph Becker, 1969
Charles P. Bourne, 1970
Pauline Atherton, 1971
Robert J. Kyle, 1972
John Sherrod, 1973
Herbert S. White, 1974
Dale Baker, 1975
Melvin S. Day, 1976

Organizational Structure. The ASIS Council, the governing body of the Society, is composed of 15 individuals: Thirteen hold office by election; the other two are ex officio.

The Council meets four times a year, in January, April, July, and during the Annual Meeting in the last quarter of the year. ASIS membership now totals nearly 4,000 individuals (including about 500 students) and more than 50 institutions.

ASIS has chartered 15 Special Interest Groups (SIGs) which provide those members with similar professional specialties the opportunity to exchange ideas and information about current and specialized developments. Special Interest Groups include the following areas:

Arts and Humanities
Automated Language Processing
Behavioral and Social Sciences
Biological and Chemical Information Systems
Classification Research
Costs, Budgeting, and Economics
Education for Information Science
Foundations of Information Science
Information Analysis Centers
Library Automation and Networks
Non-Print Media
Reprographic Technology
Selective Dissemination of Information
Technology, Information, and Society
User On-Line Interaction

The headquarters of ASIS is located at 1140 Connecticut Avenue, N.W., Suite 804, Washington, D. C. 20036. Telephone: 202-659-3644.

Technical Program. The technical and professional activities of ASIS extend from the work of the 15 Special Interest Groups and the 30 chapters to such activities on the national scale as operating the ERIC Clearinghouse on Library and Information Sciences, operating a Placement Service, and conducting a distinguished lecturer series.

Annual awards are presented for the Best Information Sciences Book, the Best Publication by an ASIS chapter or Special Interest Group, the Best Paper Published in the Journal *of the American Society for Information Science*, the Outstanding Information Sciences Movie, the Best ASIS Student Member Paper, and the Award of Merit, which is presented to a member of the profession who is deemed to have made a noteworthy contribution to the field of information science. Recipients of the Award of Merit are:

Hans Peter Luhn (posthumously), 1964
Charles P. Bourne, 1965
Mortimer Taube (posthumously), 1966
Robert A. Fairthorne, 1967
Carlos A. Cuadra, 1968
Cyril W. Cleverdon, 1970
Jerrold Orne, 1971
Phyllis Richmond, 1972
Jesse Shera, 1973
Manfred Kochen, 1974

ASIS publications include:

Journal of the American Society for Information Science
ASIS Newsletter
Handbook and Directory
Annual Review of Information Science & Technology
Cumulative Index to the Annual Review of Information Science and Technology, Volumes 1-7
Key Papers in Information Science
Directory of Educational Programs in Information Science, 1972-1973
Proceedings of the ASIS Annual Meeting

I. L. AUERBACH

AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE.

See ASCII.

ANALOG COMPUTERS

For articles on related subjects see **DIFFERENTIAL ANALYZER**; **DIGITAL COMPUTERS**; **DIGITAL TO ANALOG CONVERTERS**; **HYBRID COMPUTERS**; **NUMERICAL ANALYSIS**; **SIMULATION**; and **SPECIAL PURPOSE COMPUTERS**.

For articles on related terms see **INTERUPT**; and **REGISTER**.

BACKGROUND

The history of the analog computer goes back to antiquity, when tax maps were first reported as being used for assessments and surveying. However, this article is confined to the analog computer as it evolved in the period from World War II to the present time. (For those interested in the history of the analog computer from antiquity to World War II, the reader is referred to an excellent article by J. Roedel, 1955.)

Between World Wars I and II, much work was done in developing the mechanical differential analyzer, a close relative of the modern analog computer. Simultaneous equation solvers and harmonic analyzers of many types appeared in the 1920s and 1930s. Special computers in the form of network analyzers for the simulation of power networks appeared around 1925. The network analyzer is a passive element analog. A scale model of the particular network to be studied is made with resistors, capacitors, and inductors. The early network analyzers could be used only to investigate steady-state problems, i.e., voltage drops along lines, possible current flow in lines, etc. The more recent network analyzers can be used to investigate transient conditions during faults or switching on networks. These may be considered true general-purpose computers.

George H. Philbrick worked on an all-electronic analog computer in the mid- 1930s and is credited by many to have first used feedback amplifier theory to develop the operational amplifier (see Holst, 1971). He envisioned the analog computer as an electronic model of the system to be studied. Independently of and shortly after Philbrick's first work, the Bell Telephone Laboratories developed the M-9 Gun

Director under the impetus of the then impending World War II. The M-9 computer was a union of electronic analog computation and the mechanical differential analyzer. The first published work seems to have been handbooks accompanying the M-9 Director.

Following World War II, J. B. Russell of Columbia University brought the electronic circuitry used in the M-9 Gun Director to the attention of J. Ragazzini and others. Basing their work on the operational amplifier used in the M-9 Gun Director, Ragazzini, Randall, and Russell (1947) built an all-electronic d-c analog computer.

Immediately thereafter, several companies designed and developed analog computers for their own use and for sale to others. In 1948, Reeves Instrument Co., under a Navy contract, built the forerunner of the first commercially available analog computer.

Many companies have entered and left the analog computing field since its birth in 1948. The principal manufacturers of general purpose analog computing equipment today are Electronic Associates, Inc., Systron-Donner, Inc., Applied Dynamics Corp., Telefunken, and Hitachi.

TYPES OF ANALOG COMPUTERS

Fig. 1 shows the classification system used to characterize analog computers. The two main branches of analog computers are direct (special purpose) and indirect (general-purpose) computers, as shown in the figure.

DIRECT ANALOG COMPUTER. Direct analog computers are used in the solution of so-called field problems, e.g., conductive and convective heat transfer, fluid flow, and structures. The equations for these types of problems are partial differential equations. A "thermal analyzer" is an example of a direct analog computer that can be used in the solution of parabolic and elliptic type equations such as

$$\frac{\partial^2 \phi}{\partial x^2} = k \frac{\partial \phi}{\partial t}, \quad \frac{\partial^2 \phi}{\partial x^2} = 0$$

This type of computer has resistors and capacitors (and units that compute the fourth power of x for radiation studies). For the hyperbolic equation $\partial^2 \phi / \partial x^2 = k(\partial^2 \phi / \partial t^2)$, which describes structures, vibrating membranes, beams, etc., one might use a similar computer that has resistors, capacitors, inductors, and transformers. Both types are relatively

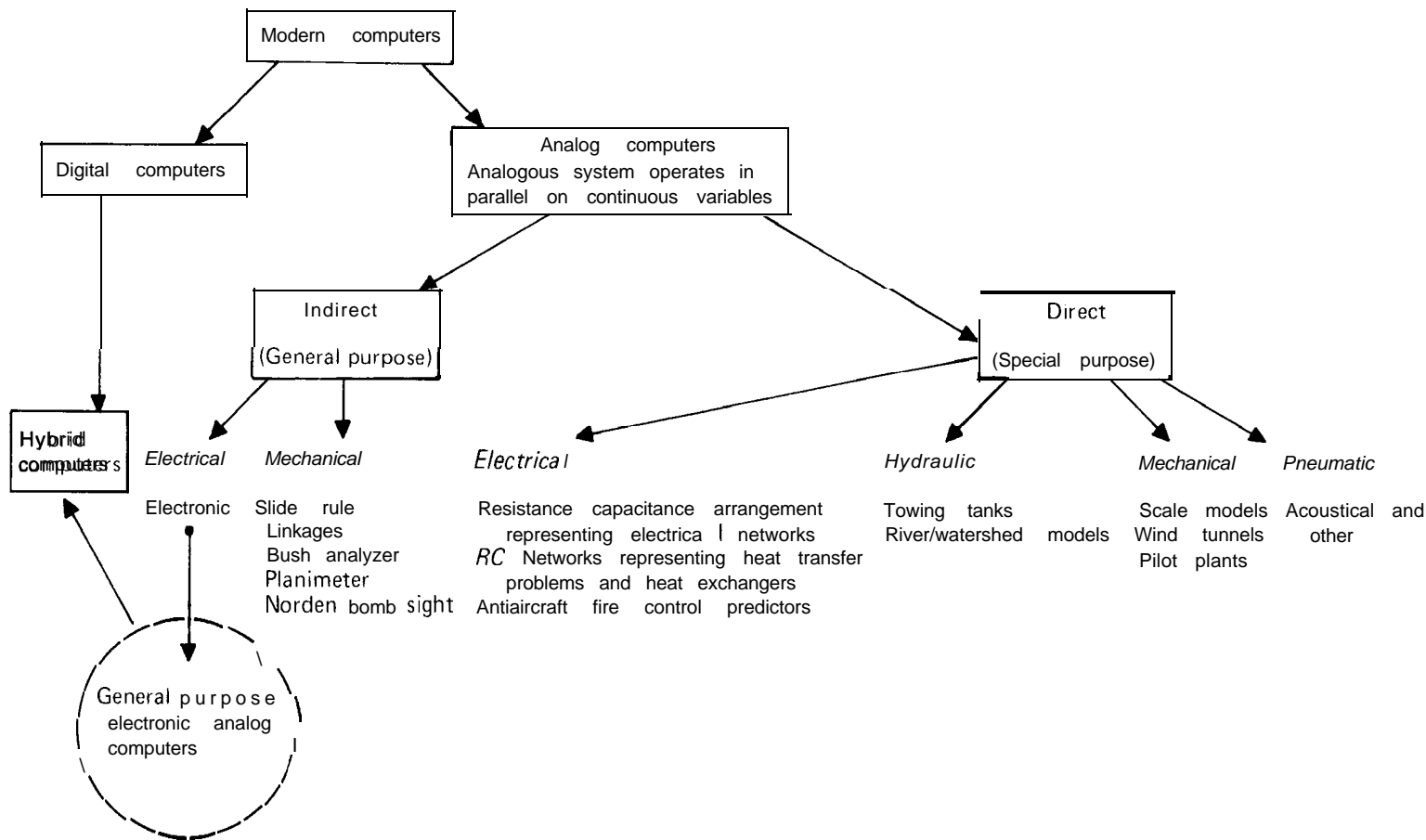
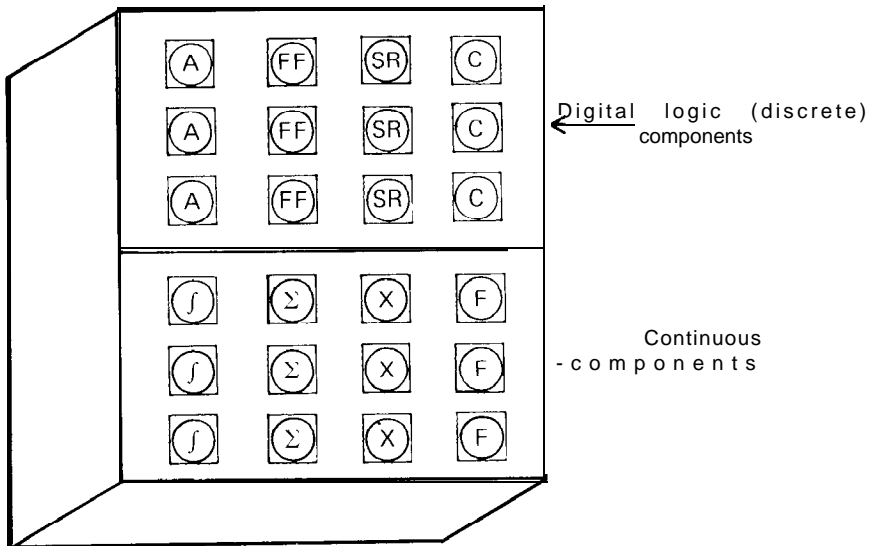
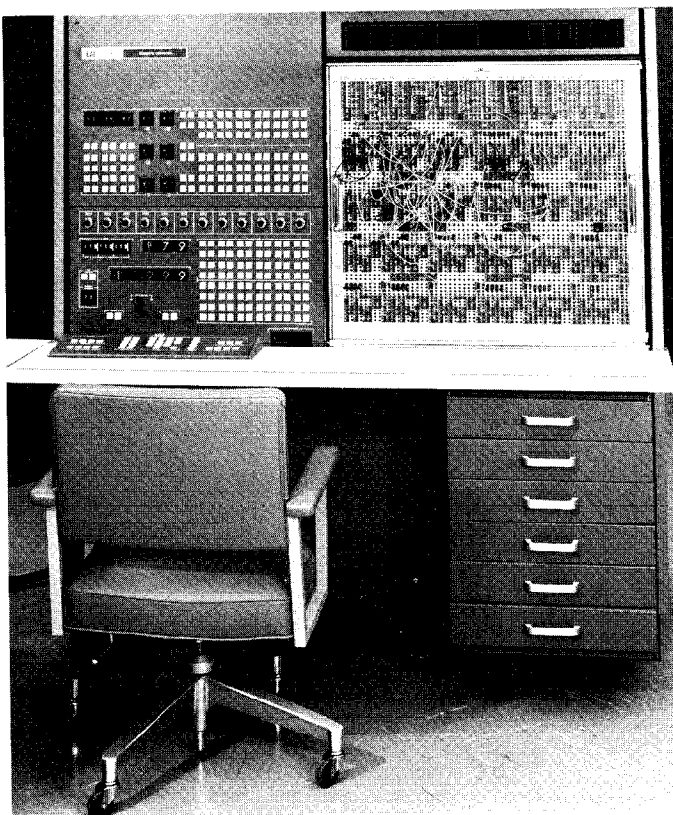


Fig. 1. Types of analog computers



(a)



(b)

Fig. 2. Hybrid computers. (a) Analog computer with continuous and discrete components organized in parallel fashion. (b) Modern analog computing system, EAX Model 680.

ANALOG COMPUTERS

special-purpose computers, and are usually referred to in the analog field as "passive analog computers."

The programming techniques of the direct analog computer and its associated problems are a subject in themselves and will not be dealt with here, except to remark that the fundamental mathematical theory of programming these partial differential equations involves finite-difference techniques.

INDIRECT ANALOG COMPUTER. The electronic differential analyzer, hereafter referred to as the "analog computer," is best suited for the solution of systems of ordinary differential equations. In mathematical terms the analog computer gives particular solutions to systems of linear or nonlinear differential equations of many variables.

COMBINED DIRECT AND INDIRECT ANALOGS. One area of study in which the analog computer is particularly useful is the so-called real-time simulation problem. In such a problem there is a requirement that the solution proceed exactly in step with real time because a man and/or equipment may be part of the overall computing loop. Such simulations allow realistic hardware testing as well as training and evaluation of complex "man-machine" systems. In these instances there is a combination of the direct analog (man is the direct analog of man, and hardware is its own best direct analog) and the indirect analog (the general-purpose analog computer).

The Modern Analog Computer. The modern analog computer consists of a large number of individual components, organized in such a manner that the inputs and outputs of these may be interconnected by a programmer-user. Fig. 2(a) shows a schematic representation of a computer as seen by a programmer. The main continuous components are integrators, represented in the figure by the symbol \int ; summers, represented by Σ ; multipliers/dividers, represented by \times ; and arbitrary function generators, represented by F. The modern analog computer also contains a number of discrete components such as "and" gates, represented in the figure by A; flip-flops, FF; shift registers, SR; and counters, C. (All these components, and other elements, will be described later in more detail.) The inputs and outputs of all elements are brought to a central patch bay, into which removable patch boards (or problem boards, prepatch panels) may be inserted. In turn, the patch boards are patched or plugged by the programmer, using patch cords (or plugs). These cords and plugs, when inserted, essentially specify the interconnections of the analog components to solve a particular problem. (For

further information on patch boards and patch cords, see Korn, 1964.) A photograph of a modern large-scale analog computer is shown in Fig. 2(b).

VOLTAGE RANGES. Most large-scale analog computers in use today have a voltage range of ± 100 volts. However, the advent of the transistor has made lower voltage ranges desirable and attractive. There are small transistor analog computers on the market today with a voltage range of ± 20 volts as well as ± 50 volts. Recently, some medium-scale computing systems have been developed with a range of ± 10 volts. The high-voltage range has the advantage of good signal-to-noise ratio and relative insensitivity to small offsets and biases that are caused by components such as diodes (in multipliers). The low-voltage range has the advantages of generally greater bandwidth (higher frequency response) and lower power requirements.

ACCURACY. The accuracy of an analog computer is usually specified by its component accuracies. The linear components in high-quality computers have errors of less than 0.01% of value or full scale, as appropriate. For example, a resistor may have an error of 0.01% of its value, but a multiplier has a fixed minimum error, which is usually stated as a percent of its full-scale output. In the latter case, the error changes with the output of the multiplier. The nonlinear components may have errors of 0.02% of full scale. Lower quality computers may have component errors as much as ten times larger than those given above.

Since a typical analog program requires the use of many computing components to obtain a solution, it is not easy to state what the overall accuracy of the solution will be. The overall accuracy depends not only on the quality (accuracy) of the components used, but also on the manner in which they are used (the program), as well as on the method of formulating the problem (analysis) for insertion in an analog computer. One can say, however, that if best practices are used throughout the programming process, the overall error of analog solutions to large problems is on the order of 0.1% to 0.5%, for the best quality computers. Since most analog solution outputs consist of recordings on X-Y plotters or strip-chart time-history recorders, the analog solution accuracy is of the same order as the accuracy of the usual output recording devices. The analog computer, however, is well matched for the job it is intended to perform, since it is used almost exclusively for the solution of engineering and scientific problems in which much of the input data is empirically determined, generally to less accuracy than the analog computer solutions thereto. For a detailed discussion

BASIC CONCEPTS

THE OPERATIONAL AMPLIFIER

1. *General.* The operational amplifier is the basic component in the analog computer. It can be used in a "summing mode" to perform the operations of inversion, summation, and multiplication by a constant. It can also be used in an "integrating mode" to integrate a voltage or the sum of a number of voltages. The change from one mode of operation to another is determined by the feedback element around the amplifier.

2. *The Fundamental Relationship.* To understand the basic operations performed by the amplifier, consider the block diagram in Fig. 3. Associated with the high-gain amplifier are the input and feedback networks, having impedances of Z_i and Z_f , respectively. Now let the voltages at the input, the output, and the amplifier grid be V_i , V_o , and E , respectively. Using Kirchoff's and Ohm's laws, we may write

$$\frac{V_o - E}{Z_f} + \frac{V_i - E}{Z_i} = i_g \quad (1)$$

where i_g is the grid current. By definition

$$\frac{V_o}{E} = -A \quad (2)$$

where $-A$ is the amplifier gain, and A is usually greater than 10^4 .

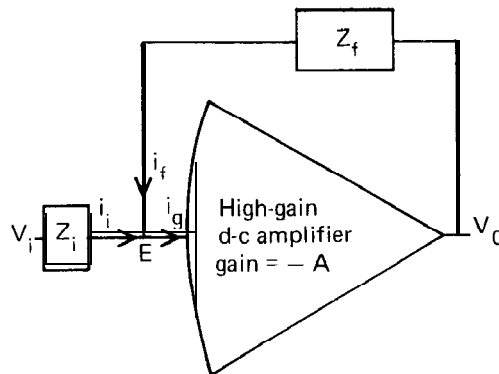


Fig. 3. Block diagram of an operational amplifier.

A further property of the high-gain amplifier is that the current i_g is at least a factor of 10^4 smaller than i_i , so that i_g can be set equal to zero; consequently, E is a voltage that is much smaller (by at

of error analysis of analog programs, see Hausner (1971).

CAPACITY. Modern self-contained analog computers may have any capacity, from the very smallest sold today (such as ten amplifiers and ten potentiometers) to the largest capacities currently being sold as single units, which have a capacity generally measured as 250–300 amplifiers, 200–300 potentiometers, 60 multipliers, 20–40 function generators, and significant quantities of digital logic devices such as comparators, flip-flops, "and" gates, "one-shots," shift registers, and counters. If a larger capacity than that available in a single unit is required in a single problem, then two or more units may be connected together to form a single, large, analog computing system. Analog computing systems containing more than 1,000 amplifiers have been successfully assembled.

A rapidly growing use of analog computers is occurring as a major portion of a hybrid computer. Hybrid computation generally enlarges the equivalent capacity of the analog part of the system by a factor of about 2. This is due to the mix of high-frequency and low-frequency parts of a problem. If a hybrid computer problem were put on an all-analog machine, it would usually require at least twice as much analog equipment as that required in the hybrid solution.

MULTISPEED OPERATION. Most modern analog computers are equipped with controls to allow instantaneous change of the speed of solution. The solution time for a large set of simultaneous nonlinear differential algebraic equations may be as short as 1 ms or as long as 100 sec by appropriate manipulation of controls. With proper programming, analog solutions can be made to last for several hours.

BASIC OPERATIONS

In an analog computer circuit for investigating the behavior of a physical system, only a few of the operational amplifiers will be used as integrators; many others will be used as "summers," "inverters," or "high-gain amplifiers." In modern equipment, about 30% of the amplifiers are able to perform all of the functions mentioned above, 45% are able to perform all operations except the integrating operation, and 25% are able to perform only the simple inverting operation. For these different arrangements the operational amplifiers are known as combination, summing, or inverting amplifiers, respectively.

ANALOG COMPUTERS

least a factor of 10^4) than either V_i or V_o , so that $E \approx 0$. Then

$$\frac{V_o}{V_i} = -\frac{Z_f}{Z_i} \quad (3)$$

This is the fundamental relationship in analog computation. The output voltage will not be affected by the internal characteristics of the amplifier; it will be governed by, and its accuracy will be dependent upon, the accuracy of the input and feedback elements.

(a) Inversion. When both Z_i and Z_f are resistors, the amplifier output will be a constant times the input voltage. If both are equal, the constant is unity and we have an inverter, as shown in Fig. 4.

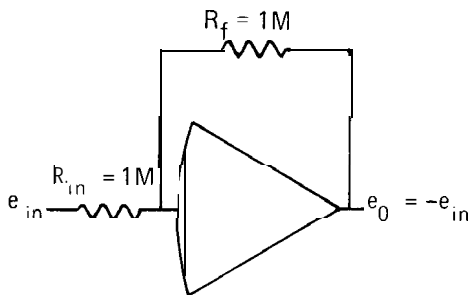


Fig. 4. The inverting amplifier (M = megohm).

To represent an inverter on computer circuit diagrams, the symbol shown in Fig. 5 is used. Fig. 5 is the "shorthand notation" for Fig. 4. Note that the number 1 at the input to the amplifier signifies a gain of 1. The change in sign is inherent with the amplifier.

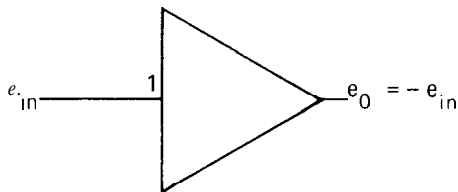


Fig. 5. The symbol for the inverting amplifier.

(b) Summation. If several input resistors are connected to a summing point SJ at the grid of an amplifier and voltages are applied to them, as shown in Fig. 6, then (owing to the fact that the grid voltage of the amplifier is effectively at zero potential) no

single input will interfere with any other input, and their effects on the output will be independent of one another. It is easily derived, then, that

$$e_o = -\left(\frac{R_f}{R_1}e_1 + \frac{R_f}{R_2}e_2 + \frac{R_f}{R_3}e_3 + \dots + \frac{R_f}{R_n}e_n\right) \quad (4)$$

The resulting output is therefore minus the sum of the input voltages, each multiplied by a constant depending upon the ratio of the resistors involved.

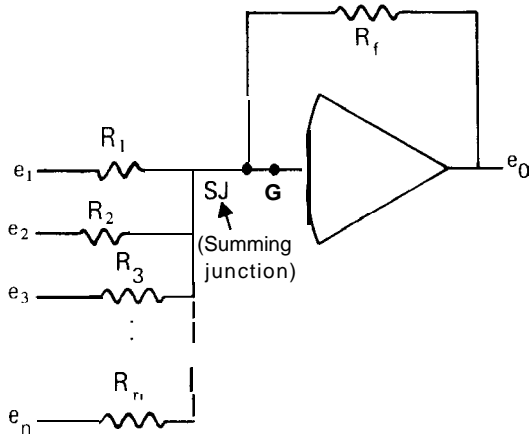


Fig. 6. The summing amplifier.

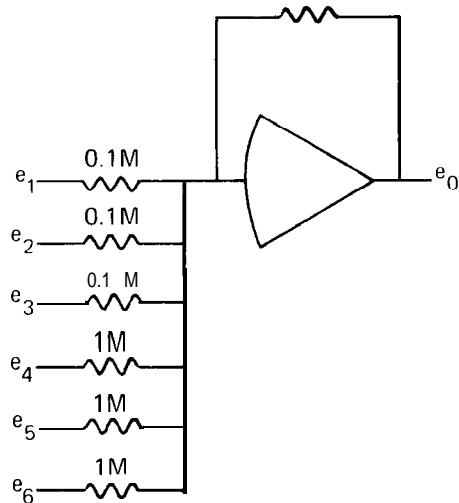


Fig. 7. Example of a summing amplifier.

From experience it was found that the most convenient values for the input resistors are 1 M (1 megohm) and 0.1M for 100-volt computers. The

resistors are correspondingly smaller on lower voltage computers. A typical summing amplifier with three 1 M and three 0.1 M resistors is shown in Fig. 7. These give gains of 1 and 10, as shown on the symbol for the summing amplifier in Fig. 8.

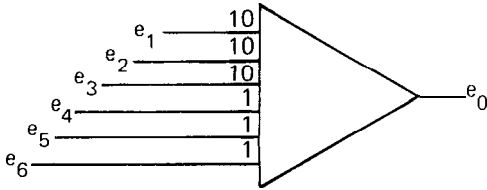


Fig. 8. Symbol for the summing amplifier.

(c) Integration with Respect to Time. Integration of an input voltage is obtained if a capacitor is substituted as the feedback component (Fig. 9).

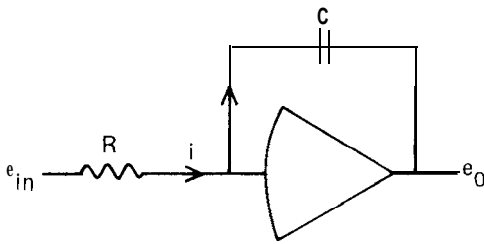


Fig. 9. Operational amplifier with capacitor feedback and resistor input.

Since the grid current i_g is zero, the current i through the input resistor R must pass through the feedback capacitor C , and will produce a potential difference between output and grid of the amplifier. Thus, in Fig. 9,

$$i = \frac{e_i}{R} \quad (5)$$

and

$$e_0 = \frac{q}{C} = \int_0^t \frac{1}{C} i dt, \quad (6)$$

where q = charge on the capacitor and C = capacitance. Thus,

$$e_0 = - \frac{1}{RC} \int_0^t e_i dt. \quad (7)$$

Alternatively, using operational notation for the impedances,

$$\frac{e_0}{e_i} = - \frac{Z_f}{Z_i} = - \frac{1}{RCp},$$

where $p = d/dt$. Therefore,

$$e_0 = - \frac{1}{RC} \int_0^t e_i dt.$$

Note that the proportionality factor RC is actually a time constant which, if we make $R = 1M$ and $C = 1\mu f$ (i.e., a time constant of 1 sec), will produce an integration rate of 1 volt/sec when e_i is equal to 1 volt.

Modern analog computers are equipped with integrators that have a variety of selectable time constants. The range of time constants normally encountered is from 10 sec (for very slow real-time solutions) to 100 μs (for very fast iterative and/or repetitive solutions).

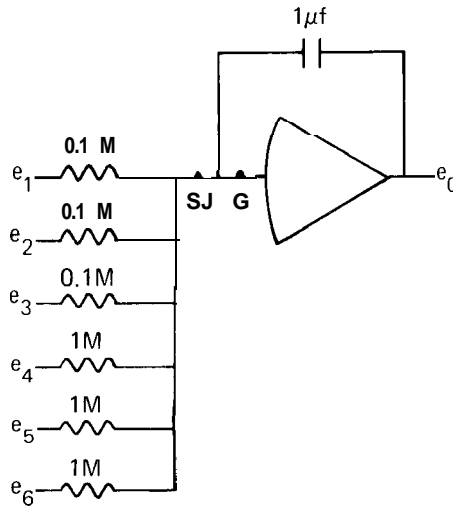


Fig. 10 A typical integrating amplifier.

Several inputs may be connected to produce the integral of the sum of a number of voltages. Figs. 10 and 11 show a typical integrating amplifier and its equivalent symbol. There is also an input terminal for inserting independent initial conditions on each integrator.

CONTROL MODES

1. Ordinary Modes

(a) Reset. This mode produces a solution at $t = 0$. All derivative terms are disconnected from the

ANALOG COMPUTERS

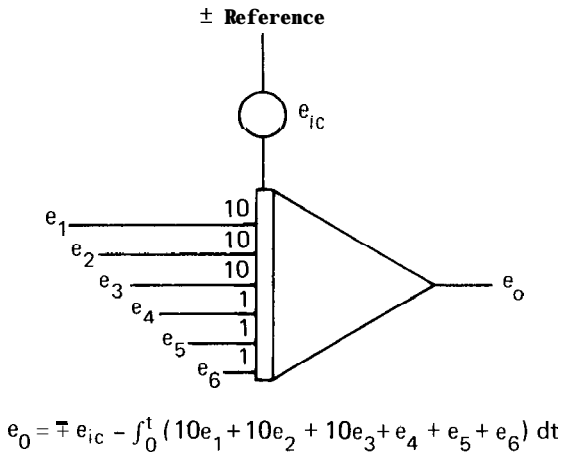


Fig. 11. The symbol for the integrating amplifier of Fig. 10, including the initial condition.

grids of the integrating amplifiers, and initial condition networks are connected by control relays or electronic gates. With $1\text{-}\mu\text{f}$ integrating capacitors, the charging time will be of the order of milliseconds, or about 1,000 times faster than previous RC feedback integrated circuits (IC). (For a thorough description of IC circuitry, see Korn and Korn, 1964.)

(b) Operate. This mode produces the time-variant solution. Derivative terms are connected to integrator grids, initial condition networks are disconnected, and capacitors associated with integrators are connected to the grids of integrator amplifiers.

(c) Hold. This mode provides a stationary solution at $t = T$ (HOLD may be selected manually by operator or selected by a computer for a previously defined value of t). Derivative terms and initial condition networks are disconnected from the integrators, capacitors remaining associated with integrators.

2. *Repetitive Operation.* In this mode all integrators are switched or cycled automatically from reset-to-operate to reset-to-operate, etc. This mode is usually associated with high solution speeds, of the order of milliseconds in duration, and with the solution displayed on an oscilloscope. When such is the case, the user will obtain the impression that a solution is obtained "instantaneously." However, it is not necessary that high solution speeds be associated with repetitive operation. All that is required is the automatic cycling of the computer between the reset and operate modes for predetermined lengths of time.

3. *Iterative Operation.* This mode may appear to be similar to the repetitive mode, but it differs from it in several respects. In iterative operation there are usually at least two, sometimes more, speeds of operation. For example, one portion of the computer may be operating at a high speed while another portion is operating at low speed. This simply requires the ability to control the integrators, either individually or in groups. The concept of "iteration" enters when the result of one speed of computation is allowed to affect the progress and/or solution of the other speed(s). This "feedback," or iterative, concept is often used in optimization, adaptive control, prediction, in the solution of certain types of partial differential equations, and in boundary value problems.

MULTIPLICATION BY A CONSTANT

1. *Potentiometers.* Multiplication by a positive constant less than unity can be achieved with a potentiometer. The most common "pots" on 100-volt computers are ten-turn, 30,000-ohm, linear, wire-wound potentiometers with one end connected to ground, as shown diagrammatically in Fig. 12. They can be used either in conjunction with the reference to obtain a fixed accurate voltage less than the reference or in conjunction with a signal voltage to multiply that voltage by any constant less than unity. For example, if $+100$ volts is applied to the high end of the pot as shown in Fig. 12, the output at the wiper will be k times 100 volts, where $k = R_1/R_T$ (neglecting the effect of external loading).

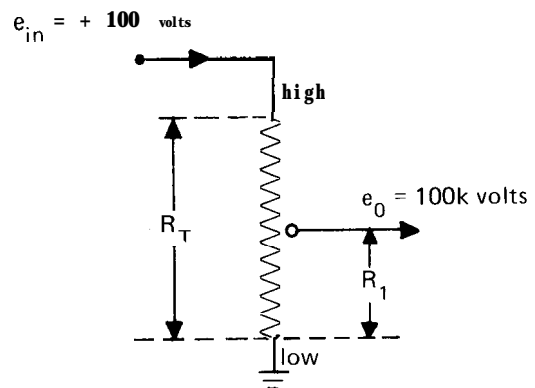


Fig. 12. Schematic of a potentiometer shown with $+100$ volts connected to the input side to give an output at the wiper of $+100k$ volts, where $k = R_1/R_T$.

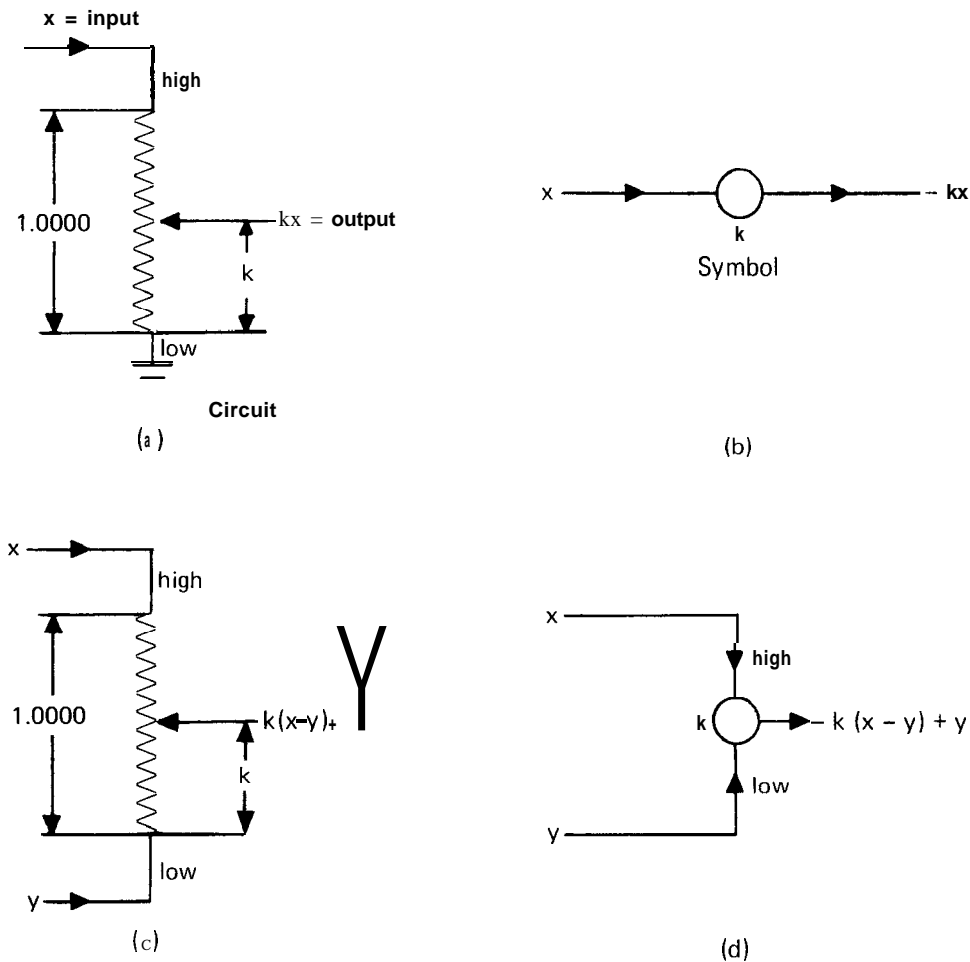


Fig. 13. Here, (b) is the symbolic representation of the attenuator shown schematically in (a); (d) is the symbol used to represent the ungrounded attenuator shown schematically in (c).

(a) The Potentiometer Symbol. Two forms of potentiometer or, as it is sometimes called, attenuator units are shown in Fig. 13; both electric circuits and analog programming symbols are shown.

(b) Pot-Set Mode. In order to set pots to their proper values under true load conditions, a special control mode called "pot set" is supplied in most analog computers. In this mode, the **SJ** (input resistors) are disconnected from the grid of the operating amplifier (see Figs. 6 and 10), and the **SJ** are grounded. Under these conditions, there will be no inputs to the amplifiers that could cause an amplifier overload while a pot is being set, for in

order to set a pot a reference must be applied to its input terminal. Note that the load seen by the pot is the same as under normal operation, for in normal operation the grid voltage E is so small that it can be considered to be the same as if it were at ground potential, the potential at which the summing junction is held during "pot set."

2. *Digital Coefficient Attenuators (DCA)*. This component is a hybridized version of a potentiometer that permits very rapid setting of coefficient values, under digital computer control, in less than $10 \mu s$. This unit is also known as a digital-to-analog multiplier (DAM) in some versions.

AMPLIFIER AND POTENTIOMETER CIRCUITS

ADDITION, SUBTRACTION, AND SIGN INVERSION

1. *Amplifiers Only.* Circuits are shown in Fig. 14.
2. *Arbitrary Gains (using pots), Including Multiplication and Division by a Constant.* Circuits providing these functions are shown in Fig. 15.
3. *Rule for High-Gain Amplifiers with Feedback.* High gain with feedback is expressed as

$$\Sigma \text{ input voltages multiplied by gains} = 0$$

This rule is true because of the high gain ($> 10^8$) of the amplifier. Assume for the moment that there is a small net voltage at the grid (even 1 mv). The high-gain amplifier would amplify this small voltage to more than full scale of the amplifier output, which

would cause the amplifier to saturate. However, in order to prevent this saturation, there must be a compensating or balancing negative feedback from the high-gain amplifier to its own input, so that for some output of the high-gain amplifier there will be an exact balance or "null" at the input, thus leading to the rule given above. This rule is illustrated in Fig. 15, where, by invoking the rule above, we have

$$ax + 10by + \frac{e_0}{K} = 0 \quad (8)$$

so that $e_0 = -K(ax + 10by)$, as indicated in Fig. 15.

4. *Gains of 0.5 and 0.10 without Pots.* Special circuits to accomplish these two functions are shown in Fig. 16.

INTEGRATION CIRCUITS. Several types of integration circuits are shown in Fig. 17.

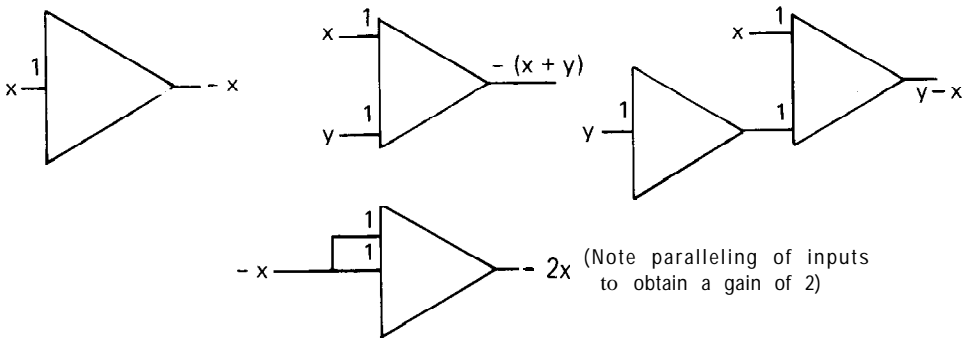


Fig. 14

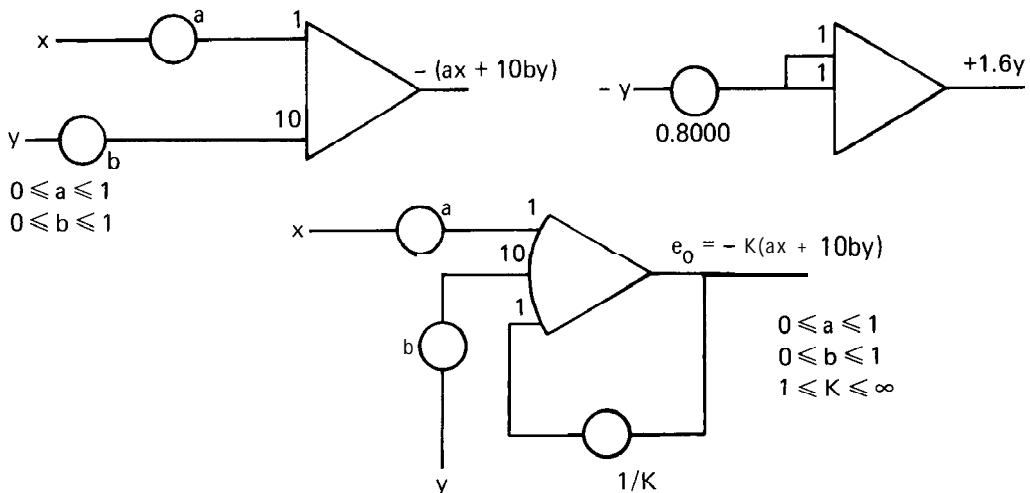


Fig. 15

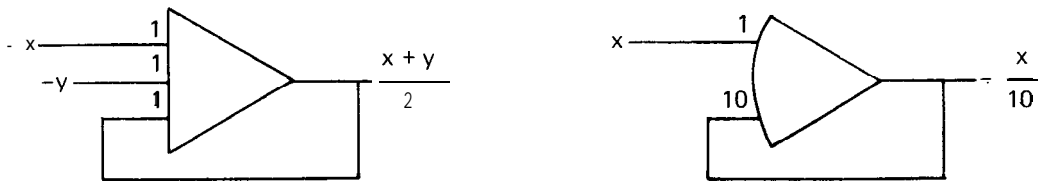


Fig. 16

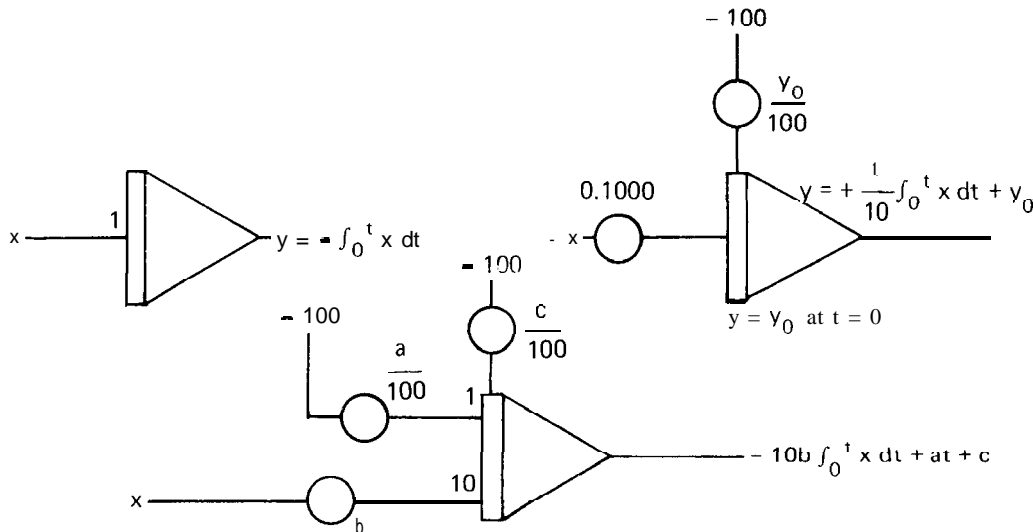


Fig. 17. Integration circuits.

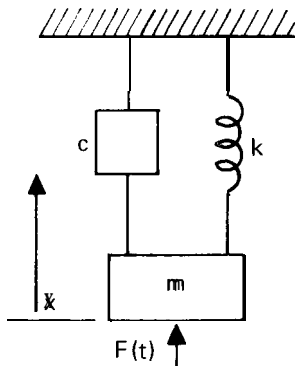


Fig. 18

THE BOOTSTRAP METHOD. Consider a mechanical system with a sinusoidal forcing function

$$F(t) = y(t) = A \sin \omega t, \quad (9)$$

where $y(t)$ is ready (of t), acting on a body of mass m which is restrained by a spring of stiffness k , and a "velocity type" damper with damping constant c . This system is shown in Fig. 18.

If x is the displacement of the body from its equilibrium position, the forces acting upon the body may be written as follows:

$$\begin{aligned} \text{External force} &= F(t) = y, \\ \text{Spring force} &= Kx, \\ \text{Damping force} &= c(\text{velocity}) = -c(dx/dt). \end{aligned}$$

The equations to be solved are

$$m \frac{d^2 x}{dt^2} + c \frac{dx}{dt} + kx = y(t) \quad (10)$$

and

Application to Linear Differential Equations. While the analog computer is most useful in solving complex, nonlinear differential equations, it is instructive to consider a linear differential equation example to learn how it is programmed.

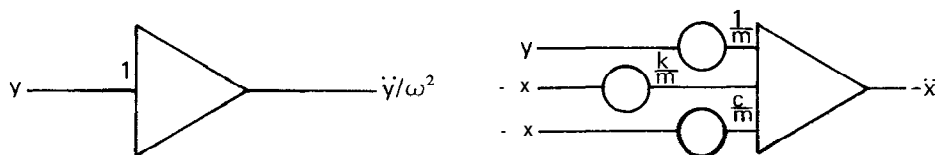


Fig. 19

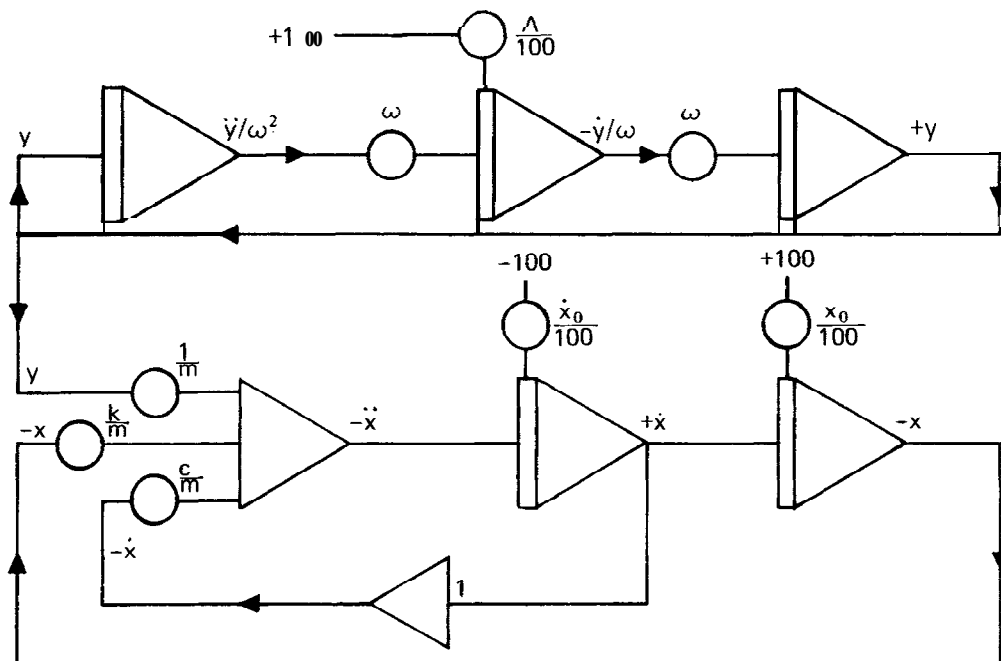


Fig. 20

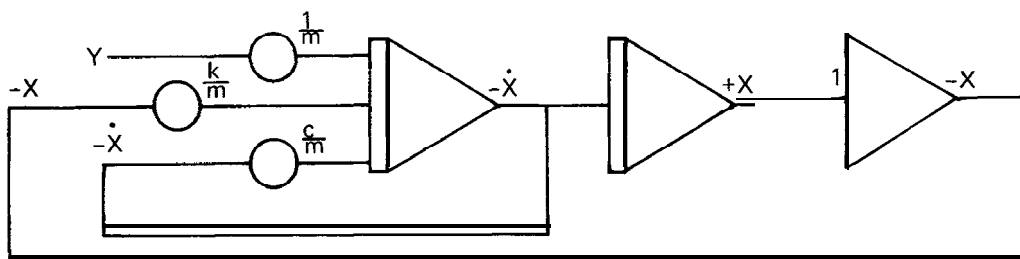


Fig. 21

$$\frac{d^2y}{dt^2} + \omega^2 y = 0 \quad (11)$$

The solution to Eq. 11 is the desired sinusoid.

The bootstrap method assumes that the terms for generating the highest-order derivatives of each variable are available. To execute the bootstrap method, the equations are rewritten in the form

$$\ddot{x} = -\frac{c}{m}\dot{x} - \frac{k}{m}x + \frac{Y}{m} \quad (12)$$

$$\frac{\ddot{y}}{\omega^2} = -y \quad (13)$$

where

$$\ddot{x} = \frac{d^2x}{dt^2}; \quad \ddot{y} = \frac{d^2y}{dt^2}; \quad \dot{x} = \frac{dx}{dt}, \text{ etc.}$$

The symbolic analog computer diagram for Eqs. 12 and 13 is shown in Fig. 19.

Using the necessary summers, integrators, inverters, and pots, the inputs to the derivatives are generated and the diagram of Fig. 19 becomes that of Fig. 20.

The initial condition for $-y/o$ is obtained from

$$-\frac{1}{\omega} \frac{d}{dt} (A \sin \omega t) \text{ at } t = 0 \quad (14)$$

An alternative method is to sum the acceleration terms for \ddot{x} directly into the \dot{x} integrator. This saves one summing amplifier, as shown in Fig. 21. The y circuit remains the same, since no saving of amplifiers would occur in that circuit.

NONLINEAR OPERATIONS

Multiplication and Division of Variables

TYPES OF MULTIPLIERS. There have been three major types of multipliers in common use on analog computers. These are the servomultiplier, the time-division multiplier, and the quarter-square multiplier. The servomultiplier (Huskey and G. A. Korn, 1962) is the oldest type and consists essentially of a servo-driven pot where the input to the servo is one variable, the input to the pot is the other variable of the product, and the output of the pot is the desired product. Because of the inherent frequency limitation of the servomechanical arrangement and the generally low reliability of such a device, these have been eliminated from modern computers.

The time-division multiplier (see Korn and Korn, 1964; Huskey and G. A. Korn, 1962) is essentially an electronic device for producing a train of rectangular pulses whose height is proportional to one variable and whose width is proportional to the second variable of the desired product. The actual product is obtained by averaging the area of the output train of pulses. The use of this type of multiplier has been restricted because of the inherent compromise that one must make between static (d-c) accuracy and wide bandwidth (dynamic accuracy). Indeed, the development of the high-accuracy, quarter-square multiplier with favorable static and dynamic accuracies has virtually eliminated the time-division multiplier. There are other types of multipliers, such as the logarithmic multiplier, the Hall effect multiplier, and triangular integration multipliers, which are used only under special circumstances and then usually not in a general-purpose computer.

THE QUARTER-SQUARE MULTIPLIER. This all-electronic multiplier gives the best accuracy, reliability, and frequency response of any general-purpose multiplier. Its fundamental operation is derived from the relation

$$\frac{1}{4}(X + Y)^2 - \frac{1}{4}(X - Y)^2 = XY. \quad (15)$$

For example, quarter-square multiplication could be mechanized as shown in Fig. 22. The boxes marked **FG** are function generators (described in the next section), which here have the property of producing the square of the input variable.

General-purpose analog computers have quarter-square multipliers with fixed squaring networks that can be used for either one product or two squares. Here we adopt the convention that the multiplier has all the necessary hardware and therefore can be regarded as a "black box." The symbol for diagrams is shown in Fig. 23.

SQUARING. Squaring is accomplished by connecting the same variable to both inputs of a multiplier, as shown in Fig. 24.

DIVISION AND SQUARE ROOT BY USE OF IMPLICIT ARITHMETIC. A nonlinear component may be used in the feedback loop around high-gain amplifiers to perform the inverse of the operation that the component performs in the forward loop configuration. The most frequent use of this technique is in division and square root circuits with multipliers.

1. Square *Root*. Refer to Fig. 25. Let

$$\epsilon = X - Z^2 \quad (16)$$

and assume that, for a high-gain amplifier, the

ANALOG COMPUTERS

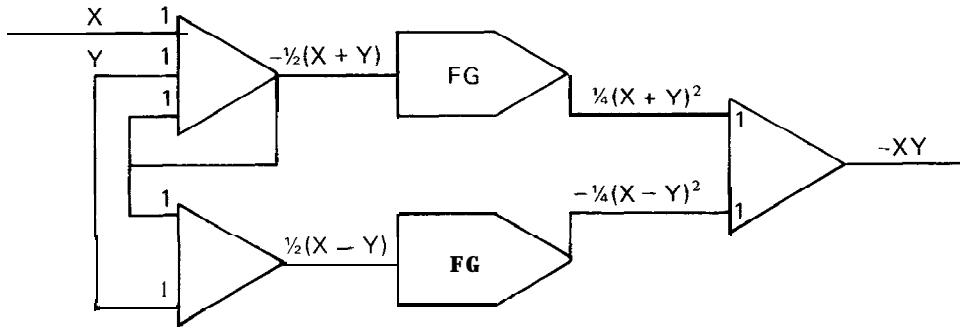


Fig. 22 Quarter-square multiplication

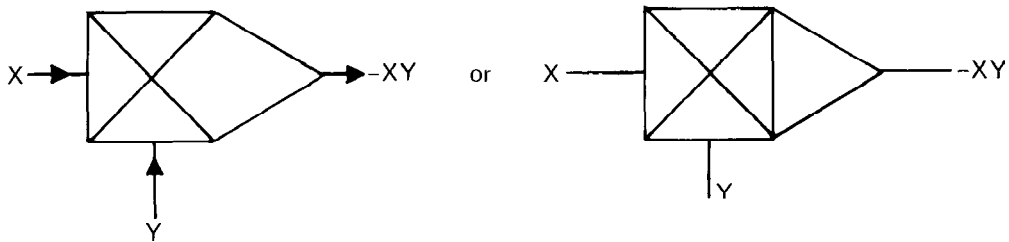


Fig. 23. Multiplier symbol (note the sign inversion).

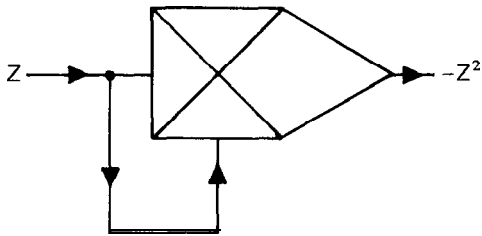


Fig. 24. Squaring circuit.

output is related to the input grid voltage

$$Z = A\epsilon \quad (17)$$

where $A > 10^8$. Eliminating ϵ ,

$$X - Z^2 = \frac{Z}{A} \simeq 0. \quad (18)$$

Therefore,

$$Z = (X)^{1/2}. \quad (19)$$

For stability, the feedback loop must have an odd number of inversions of the signal so that the sum of the currents through the input resistors to the amplifier grid is equal to zero (ϵ). It is this rule that allows the determination of the sign of the output, which would otherwise be indeterminate.

Note that a squaring device has the property of acting as a sign changer for only one sign of the input variable. In analog multipliers there is usually a built-in sign inversion, as described previously under "Amplifier and Potentiometer Circuits," so that analog squarers act as sign inverters for positive

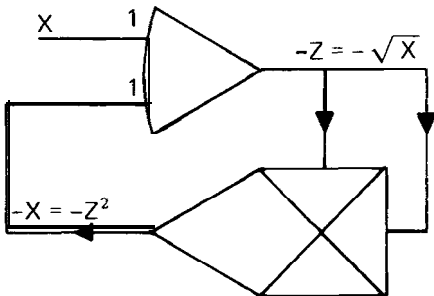


Fig. 25. Square root circuit ($X > 0$).

inputs only. Consequently, in the square root circuit the squarer counts for zero inversions, since when Z is negative the output $-Z^2$ is also negative. The one inversion in the circuit is the high-gain amplifier producing $-Z = -(X)^{1/2}$.

Note also that the circuit is stable only for $X > 0$. For values of $X < 0$, an additional inverter must be placed in the feedback loop, and the output of the high-gain amplifier becomes $+Z = (-X)^{1/2}$.

Since modern analog computers have provision for automatically converting a multiplier to a square root circuit, a convenient symbol to use is shown in Fig. 26.

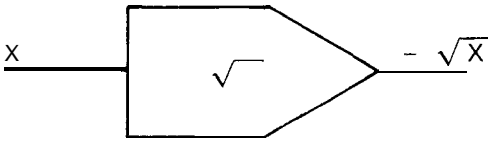


Fig. 26. Symbol for square root circuit.

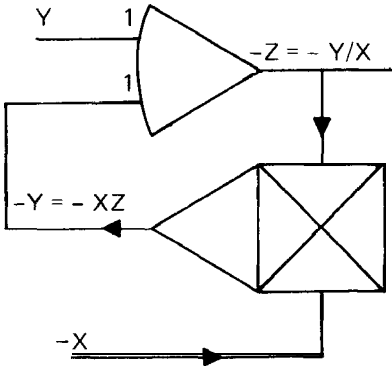


Fig. 27. Division circuit ($X > 0$).

2. Division Circuit. Similarly, for division (Fig. 27), let

$$\epsilon = Y - XZ, \quad Z = A\epsilon, \\ Y - XZ = Z/A \approx 0, \quad Z = Y/X.$$

Note that X must be positive but that Y can be of either sign. Also note that the negative of X must be brought to the multiplier terminal in order to satisfy the stability rule described above for the square root circuit.

Since modern analog computers have provision for automatically converting a multiplier to a divider, a convenient symbol to use is shown in Fig. 28.

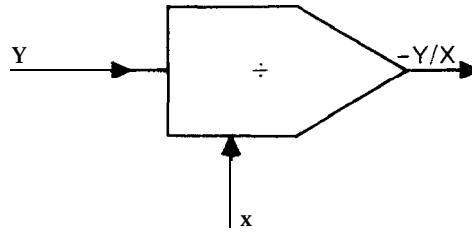


Fig. 28. Symbol for divider.

SPECIAL MULTIPLIER HOOKUPS. Some modern multipliers have provisions for obtaining special sign-sensitive squares and square roots, which are important in fluid flow phenomena. As an example, take the case of the flow of fluid through an orifice, which is proportional to the square root of the pressure drop across the orifice. If the reverse flow is to take place, it is necessary to implement the equation

$$Q = \text{sign}(\Delta P) (\Delta P)^{1/2} \quad (20)$$

Similarly, drag forces acting on bodies moving through fluids are generally proportional to the square of the relative velocity between body and fluid, and are opposite in sign to the direction of motion. It is necessary to implement the equation

$$C_{\text{drag}} = -\text{sign}(V) \cdot V^2. \quad (21)$$

By a simple patch change on modern analog computers, the two operations exemplified by Eqs. 20 and 21 are directly implemented without requiring any special logic-switching operations. Since these are direct analog outputs, convenient symbols may be used as shown in Fig. 29. Note that the two special multiplier hookups in Fig. 29 apply only to squaring and square rooting.

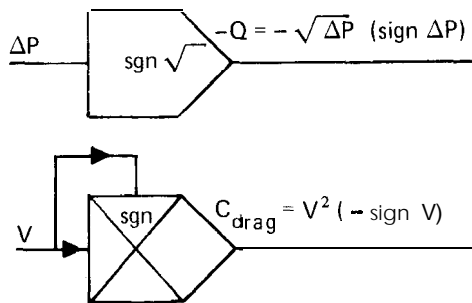


Fig. 29. Other convenient nonlinear programming symbols.

Function Generators. There are two types of function generators commonly in use today, diode function generators (DFG), and digitally controlled function generators (DCFG). These are used to insert, or input, arbitrary functions of one variable, using a piecewise linear approximation on from 10 to 20 arbitrarily spaced points in the independent variable.

THE DIODE FUNCTION GENERATOR. This component, which has been available since 1955, accomplishes the FG operation by the circuitry shown in Fig. 30, using the techniques discussed in the next section, "Simulation of Discontinuities."

By modifying the dead space circuit (refer to the later discussion "Simulation Discontinuities"), having both signs of the input and reference voltages available, one can choose "breakpoints and slopes" at will, as shown in Fig. 31.

The circuit works as follows: If X is positive, the lower diode is biased beyond cutoff (rendered non-conducting = open circuit) so that only the upper

diode circuit can contribute. In the region $0 \leq X \leq \text{b.p.}$ (where b.p. is the breakpoint setting of the upper b.p. pot), the upper diode is also biased beyond cutoff so that there is no input to the Y amplifier (both input diodes are on open circuit). This is also shown in the characteristic graph of Y versus X in Fig. 31(b); i.e., there is no output Y between zero and the breakpoint. Now, when X is positive and greater than the upper breakpoint, the output of the upper b.p. pot will be positive, increasing linearly with X from a zero value when X is at the breakpoint value; see Fig. 13(d). The slope of the output characteristic will be determined by the slope pot. Note that the input to the Y amplifier is positive, thus creating the negative output Y as shown on the characteristic graph. In a similar manner, it can be shown that when X is negative, the upper diode is always biased beyond cutoff, and that the lower diode will also be cutoff for $-\text{b.p.} \leq X \leq 0$, where $-\text{b.p.}$ is the breakpoint setting of the lower b.p. pot. At this point the analysis of the

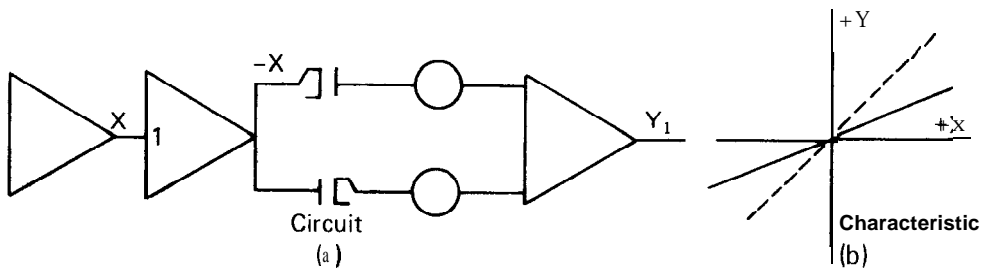


Fig. 30. Diode circuit for output slope change at origin.

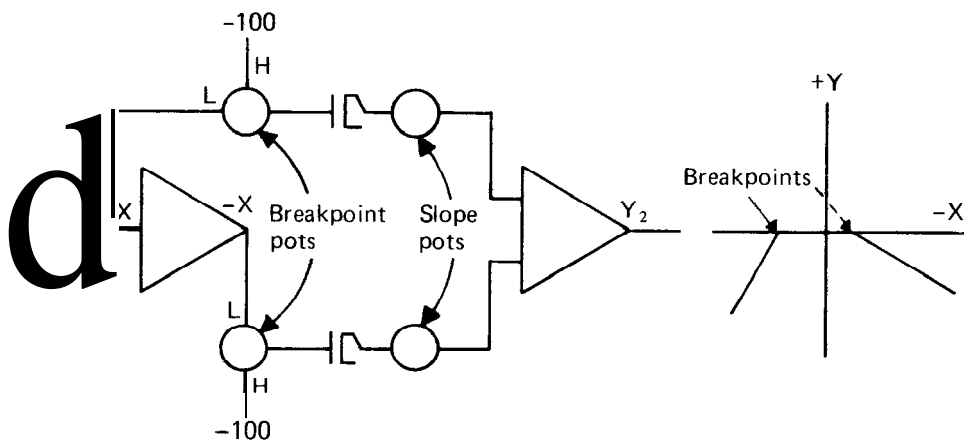


fig. 31. Diode circuit for output slope changes away from the origin (see Fig. 13(d) for three-terminal pot).

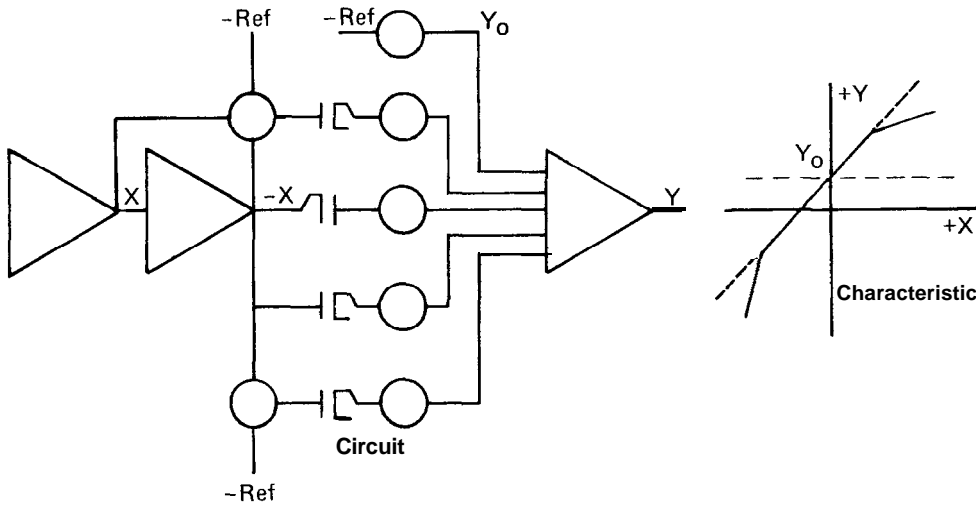


Fig. 32. Complete DFG circuits, including bias pot Y_0 .

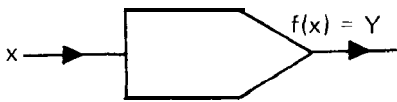


Fig. 33 Function generator symbol.

lower circuit is identical to the upper circuit, since $-X$, the input to the lower circuit, is now a positive voltage.

To obtain positive output values Y [in the upper two quadrants of Fig. 31(b)], it is only necessary to reverse the polarity of the diode connection while at the same time changing the polarity of the reference voltage on the corresponding breakpoint pot.

By combining the two circuits in Figs. 30–31, we have a circuit that produces an output function $Y(X)$; i.e., a superposition of the two functions. A coefficient pot from the negative reference voltage is added so that $Y_0 \neq 0$ (see Fig. 32). By extension of this technique, straightline segment approximations are obtained for a wide variety of arbitrary functions. The symbol for an arbitrary function generator of one variable is shown in Fig. 33.

THE DIGITALLY CONTROLLED FUNCTION GENERATOR. The digitally controlled function generator (DCFG) is a hybrid computing device, now supplied as a fully self-contained unit in existing analog computers. It consists of a small, high-speed core memory (to contain the function data points) and multiplying digital-to-analog converters, organized as shown in Fig. 34.

The function $f(x)$, to be generated, is computed by a linear interpolation between function values $f(x_i)$ and $f(x_{i+1})$, where x_i is a general “breakpoint” value. The number of breakpoints is typically 16, and they can be unequally spaced.

If x is the independent (input) variable, then

$$Ax = \frac{x - x_i}{x_{i+1} - x_i} \quad (22)$$

is the normalized value of x in the interval $[x_i, x_{i+1}]$. The equation used to generate $f(x)$ is then

$$f(x) = \Delta x f(x_{i+1}) + (1 - \Delta x)f(x_i). \quad (23)$$

In Fig. 34 the independent variable $-x$ (at the lower left) is summed with a digital-to-analog converter (DAC) containing x_i , and is divided by a multiplying DAC (MDAC) containing $x_{i+1} - x_i$. The output of this circuit is Ax , defined in Eq. 22. The output Ax is subtracted from the reference, forming $1 - Ax$, and both Ax and $1 - Ax$ are fed to the MDAC (at top of figure) containing $f(x_{i+1})$ and $f(x_i)$, respectively, thus forming the output $f(x)$.

The control and logic for changing the digital data in the DAC and MDAC are shown in the lower right of Fig. 34. Here, Ax enters two comparators (see later section, “The Analog Comparator”), one sensing when Ax is less than zero, the other sensing when Ax is greater than the reference. Both logic outputs are connected to priority interrupt lines in the processor containing the digital data $f(x_i)$ and x_i , $i = 1$ to n .

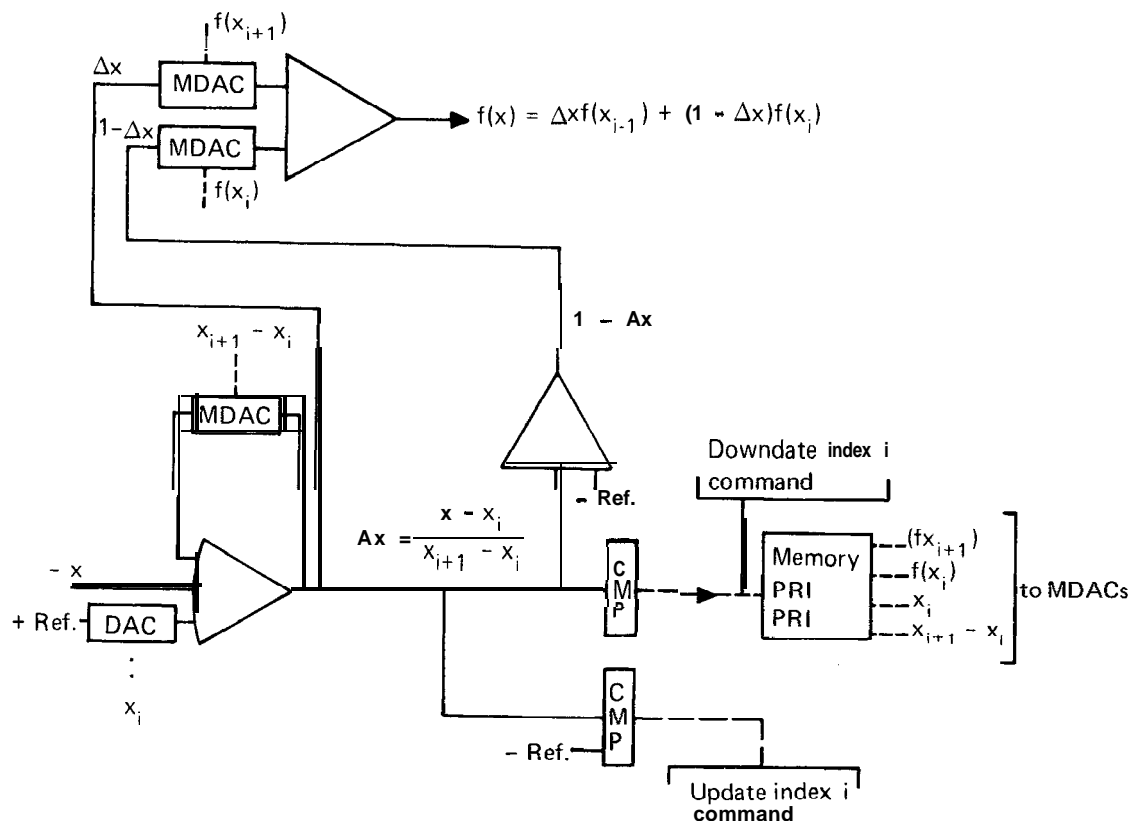


Fig. 34. Digitally controlled function generator,

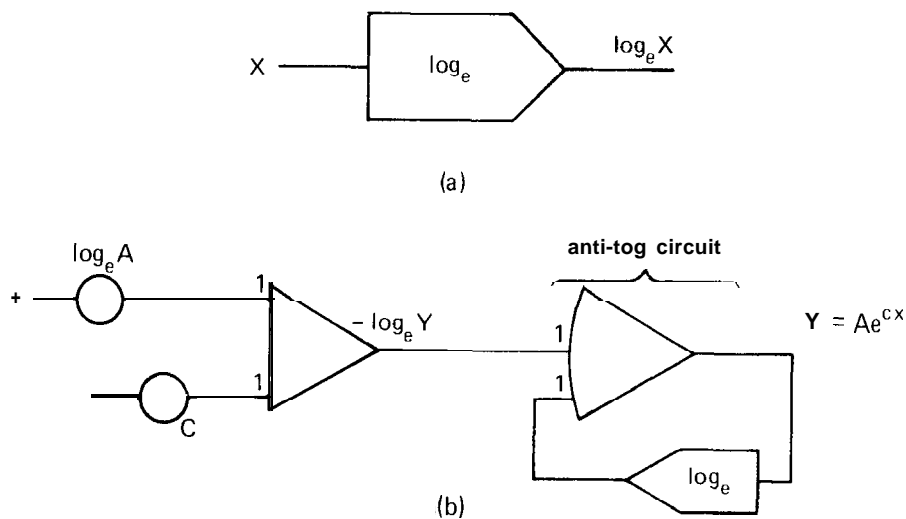


Fig. 35. Fixed function generator for generating the natural logarithm of a variable.
(a) Symbol. (b) Circuit.

One comparator triggers a **downdate** of the index i and the other triggers an update of the index i . Whenever a trigger occurs, the appropriate values of x_i , $x_{i+1} = x_i$, $f(x_i)$, and $f(x_{i+1})$ are transferred within a few memory-cycle times to the appropriate DAC and MDAC, thus allowing the circuit generating $f(x)$ to be correct in all intervals $x_{i+1} - x_i$.

SPECIAL FUNCTION GENERATORS (FIXED-FUNCTION GENERATORS). Certain functions such as exponentials, sines and cosines, squares, and cubes recur so often in engineering and scientific studies that it has been found useful to build fixed-function generators for these operations.

1. *Exponential Log Generator.* Perhaps the most flexible method for generating an exponential is to use a fixed-function generator from which any exponential can be generated. The symbol for such a device is shown in Fig. 35. Using this device, it is

possible to generate any exponential by employing the logarithm generator in the feedback of a high-gain amplifier, thus obtaining the inverse operation (or antilog). (This is analogous to using a multiplier in the feedback of a high-gain amplifier to obtain division.)

For example, to generate the exponential $A e^{cx}$, where A and c are constants and x is a variable, let $y = A e^{cx}$. Then

$$\log_e y = \log_e A + cx. \quad (24)$$

The circuit for forming the $\log_e y$ from Eq. 24 is shown in Fig. 35(b). Inserting this sum into a high-gain amplifier, which has a log generator in its feedback path, will take the antilog of the input, thus producing the desired output y .

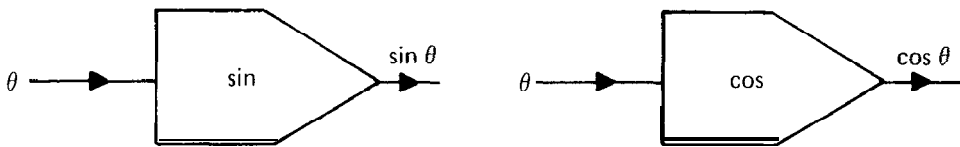


Fig. 36. The sin/cos generators.

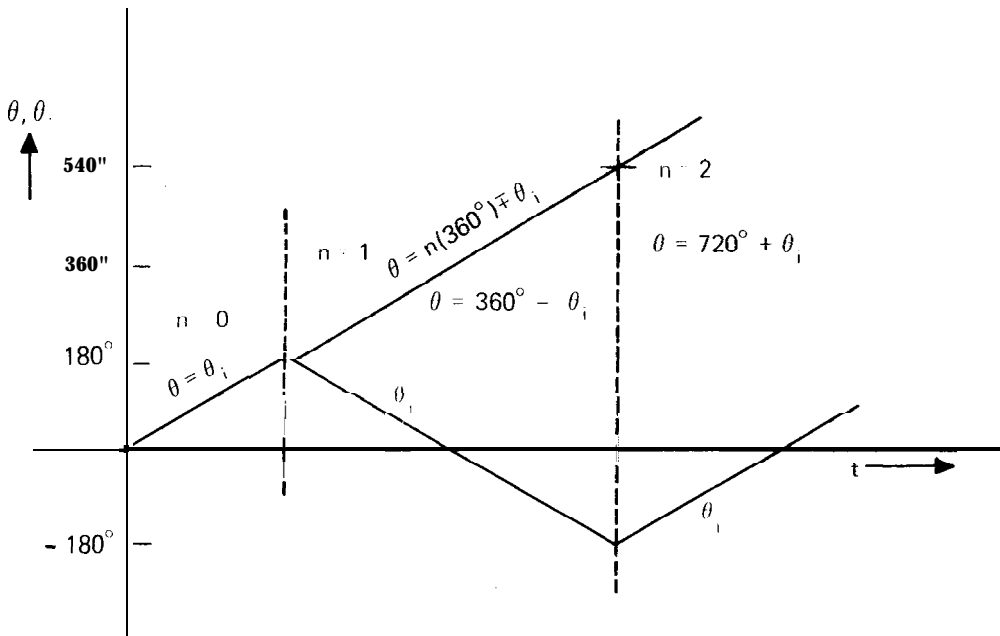


Fig. 37. Time history showing θ and θ_i with switching occurring at $\theta = 180^\circ$ and $\theta = 540^\circ$.

2. The Resolver (Sine-Cosine Generator). A resolver is actually a combination of computing elements, including provisions for generating $\sin \theta$ and $\cos \theta$, given θ as an input, and also allowing for the multiplication of both $\sin \theta$ and $\cos \theta$ by any other variable V , thereby generating $V \cos \theta$. This device is an outgrowth of servomultiplying technology, wherein it was a relatively simple matter to change a linear pot to a sine or cosine pot (padded-pot technique) and (by applying $\pm V$ to the endpoints of the padded pot) to obtain $V \sin \theta$ and $V \cos \theta$. Modern computers, however, usually have a fixed (electronic) function generator to generate either the sine or the cosine function. This is shown symbolically in Fig. 36.

It is, of course, still possible to combine the preceding operation with electronic multipliers to obtain $V \sin \theta$ and $V \cos \theta$. If one merely has a sine or cosine generator, then it is termed a "sinusoid" generator to include both functions (since it requires only a single patching change to obtain either function). If the sinusoid generators are intimately packaged with the multipliers to allow direct generation of $V \sin \theta$ and $V \cos \theta$, given V and θ as inputs, then the package is called a "resolver."

(a) Rate Resolver. A rate resolver allows the insertion of $\dot{\theta}$, instead of θ , into a resolver input

terminal, and $\sin \theta$ and $\cos \theta$ will be automatically produced. This is simply accomplished by inclusion of an integrator within the resolver package, which will integrate $\dot{\theta}$ and produce θ .

(b) Continuous Rate Resolver. The normal allowed range of input to the sinusoid generator (SG) is ± 180 deg. If θ should go larger than this—as, for example, in continuous rolling and/or tumbling—then a switch is incorporated on the rate input, which changes the sign of the θ input to the resolver integrator whenever $|\theta_i|$ reaches 180 deg. At the same time, the sign of $\sin \theta$ is changed. This follows from the relations

$$\begin{aligned} \theta &= n(360^\circ) \pm \theta_i, & |\theta_i| < 180^\circ, \\ \sin \theta &= \pm \sin \theta_i, & \cos \theta &= \cos \theta_i, \end{aligned}$$

$$\text{Input to SG} = \theta_i,$$

where $n = \pm 0, 1, 2$, etc.

The \pm signs depend upon whether n is odd or even and whether θ is increasing or decreasing. A time history of θ and θ_i for increasing θ is shown in Fig. 37.

In particular, if $n = 1$ and $\theta = 360^\circ - \theta_i$ and $\sin \theta = -\sin \theta_i$, then a sign change must occur at the output of the sine generator for odd n . Similarly, when $\theta = 360^\circ - \theta_i$, then $\cos \theta = \cos \theta_i$, which is correct for all n . The circuit is shown in Fig. 38.

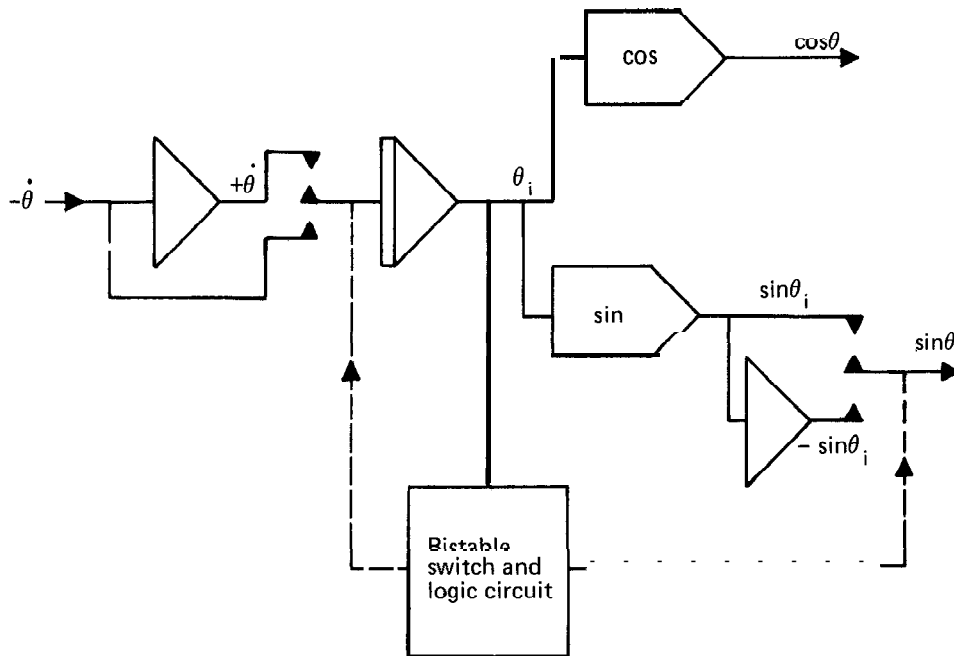


Fig. 38. Rate resolver and continuous resolver (multipliers not shown).

(c) Polar Resolution. The object here is, given the x and y components of a vector (or a complex variable), to find R , the magnitude of the vector, and θ , the angle that the vector makes with the X-axis. This is accomplished by forming the error equation $\epsilon = x \sin \theta - y \cos \theta$, which, as can be seen from the geometry of the relationships among x , y , and θ as shown in Fig. 39, is zero only when θ is the correct angle. In these circumstances—i.e., when an implicit algebraic relationship must be satisfied by a dependent variable θ —given the independent variables x

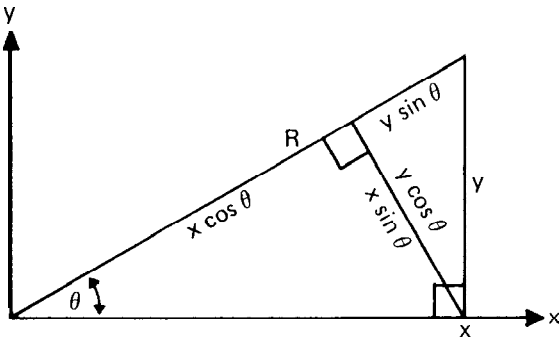


Fig. 39 Polar resolution geometry.

and y , a mathematical method exists, called the “method of steepest descent” (see Hausner, 1971), which defines a stable formula for the generation of the time derivative of the dependent variable as follows:

$$\frac{d\theta}{dt} = -k\epsilon \frac{\partial \epsilon}{\partial \theta}, \quad (25)$$

where ϵ is as defined above and k is an arbitrary constant. From the definition of ϵ we derive

$$\frac{\partial \epsilon}{\partial \theta} = x \cos \theta + y \sin \theta. \quad (26)$$

From the geometry of Fig. 39, we see that $x \cos \theta + y \sin \theta = R$, so that $\partial \epsilon / \partial \theta = R$. Substituting the last expression into the original equation for $d\theta/dt$, we obtain $d\theta/dt = -k\epsilon R$. The circuit for obtaining R and θ from x and y is shown in Fig. 40.

(d) Polar Resolution Circuit. For fastest response in this circuit, k should be made as large as loop stability will permit. This is usually a value between 1,000 and 10,000. Such large gains are obtained by using small capacitors for the integrator feedback ($0.01 \mu\text{f}$ or smaller).

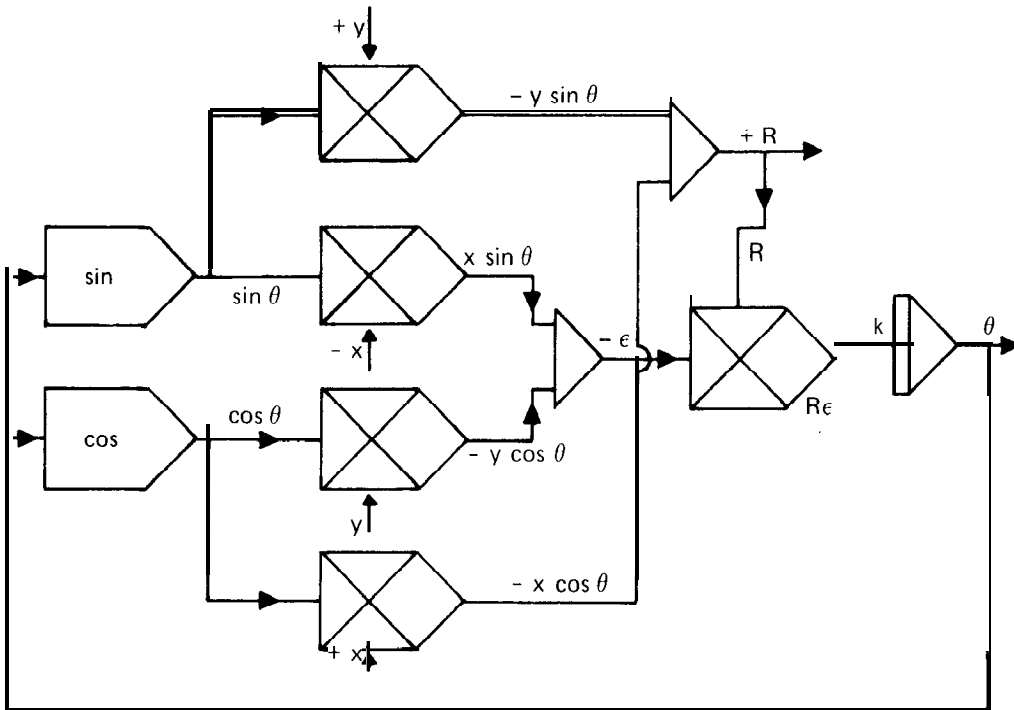


Fig. 40. Polar resolution circuit.

ANALOG COMPUTERS

3. X^3 and X^4 Generators. These are similar in operation to the previously discussed special generators, differing only in the output function. The programming symbols are shown in Fig. 41.

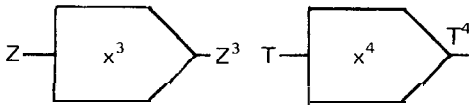


Fig. 41. Programming symbols for X^3 and X^4 units.

The X^4 generator is particularly useful in heat radiation studies.

FUNCTIONS OF MORE THAN ONE VARIABLE. Most analog programmers must resort to some mathematical juggling or simplification of functions in order to be able to insert multivariable functions on an analog computer. For example, a function $F(x,y)$, may sometimes be expressed as the sum or product of two functions of one variable, such as

$$f_1(x,y) = g_1(x) + h_1(y),$$

or

$$f_2(x,y) = g_2(x)h_2(y) + \text{similar terms.}$$

More details on purely analog techniques for multivariable function generation may be found in Hausner (1971), and Huskey and G. A. Korn (1962). There exists a general method for handling the generation of functions of any arbitrary number of variables, but it requires a hybrid computer. This is essentially an extension of the method described for the DCFG.

Simulation of Discontinuities.

Discontinuities (such as limit stops, rate limits, dead zones, sudden changes of gain, and opening or closing of circuits) are programmed on the analog computer by means of diodes and/or electronic gates. A diode may be regarded as a voltage-sensitive on-off switch. As a first approximation we consider the circuit to be closed (conducting), if the plate is positive with respect to the cathode, and open (nonconducting) if

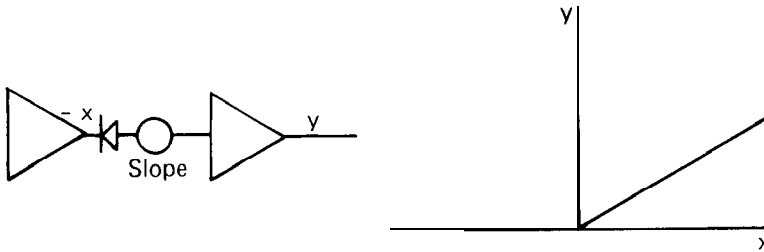


Fig. 42. Origin discontinuity circuit.

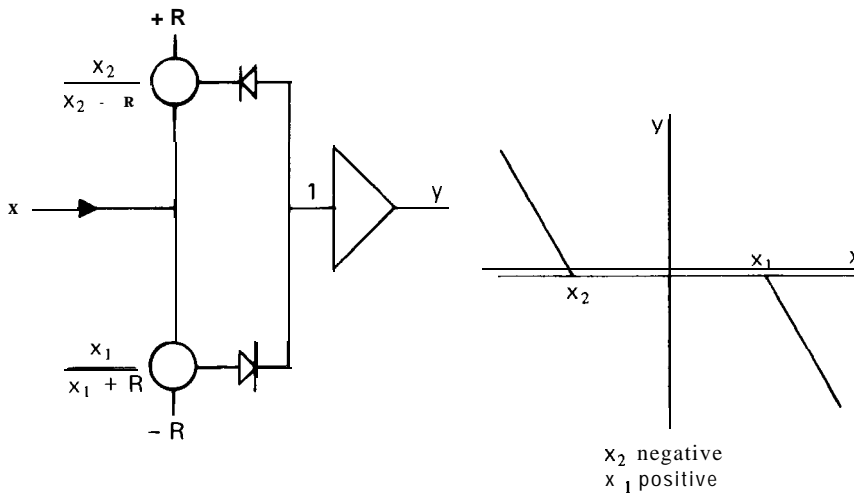


Fig. 43. Dead-space circuit.

the plate is negative with respect to the cathode. A simple circuit for introducing a discontinuity at the origin is shown in Fig. 42.

In the circuit shown in Fig. 42, $-X$ is connected to the cathode of the diode and the plate is connected to a pot. When $-X$ is negative, the cathode of the diode is negative with respect to the plate, so the diode conducts and produces a positive output through the inversion of the Y-amplifier. When $-X$ is positive, the diode is rendered in the nonconducting state, and $Y = 0$. The circuit characteristic is shown to the right of the circuit diagram. This circuit is also called a nonnegative limiter (i.e., Y is constrained to positive values only). By reversing the diode, one can make a nonpositive limiter. The circuit in Fig. 42 can be considered to have a breakpoint at zero (a discontinuity in the derivative of the output occurs when $X = 0$). The discontinuity in the output can be made to occur at any arbitrary value of X , as in the "dead-space" circuit shown in Fig. 43; see Fig. 13(d). Notice that the discontinuity occurs at other than $X = 0$.

A group of common diode circuits with their input-output characteristics is shown in Fig. 44.

DIGITAL LOGIC OPERATIONS

The Analog Comparator. The analog comparator has been a fundamental component of the analog computer from its inception. In the past it was intimately associated with a relay such that the comparator output drove the relay arm to one of two sets of contacts. Actually, the analog comparator is a true hybrid device, since it accepts analog inputs (usually two) and produces a digital logic level output (either a binary "1" or a binary "0"). The symbol is shown in Fig. 45.

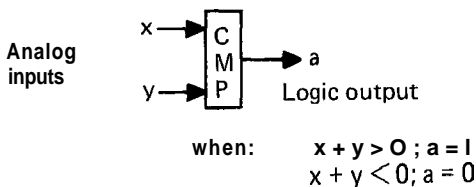


Fig. 45. Analog comparator symbol.

If one of the two analog inputs is a constant voltage (as, for example, a reference voltage multiplied by a constant coefficient), then the output of

the comparator shows when the other variable is greater than or less than a particular constant value.

The output of the comparator can be used to control the analog computer to drive electronic gates or as inputs to other digital logic components, to sense lines, control lines, interrupt lines, or priority interrupt lines of digital computers.

General-Purpose Digital Logic Modules. It may seem strange to include a section on true general-purpose digital logic components with material on analog computers, but analog programmers have always made use of digital logic in the normal course of obtaining a solution to a problem.

Not many years ago, general-purpose digital logic modules were not available, so the manufacturers of analog equipment did not supply such modules. The programmer, however, by using comparators, relays, diodes, limiters, and amplifiers, was usually able to "simulate" digital logic. This "logic" was asynchronous, and operated in parallel, so that outputs of all logic components were available to the programmer at all times. At present, analog manufacturers include a good supply of digital logic modules as part of the normal computing complement of the analog computer. These modules are patched one to another, just as analog components are, and operate in parallel and simultaneously, as analog components do. In view of the last statement, one may consider such logic modules to be discrete analog components.

The most common types of logic modules used with analog computers are flip-flops, "and" gates, "or" gates, "one shots" (or "pulsers," or "time delays," or "monostables"), and combinations of these elements to produce "exclusive or" circuits (or "modulo 2 adder," or "ring sum"), up-and-down counters, and shift registers.

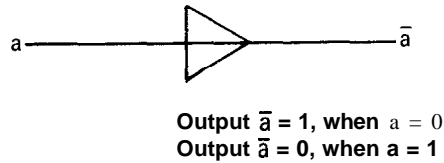


Fig. 46. Digital inverter symbol.

Associated with the logic modules is the concept of a digital inverter, shown in Fig. 46. A table of symbols and functions of common digital logic modules is shown in Fig. 47.

ANALOG COMPUTERS

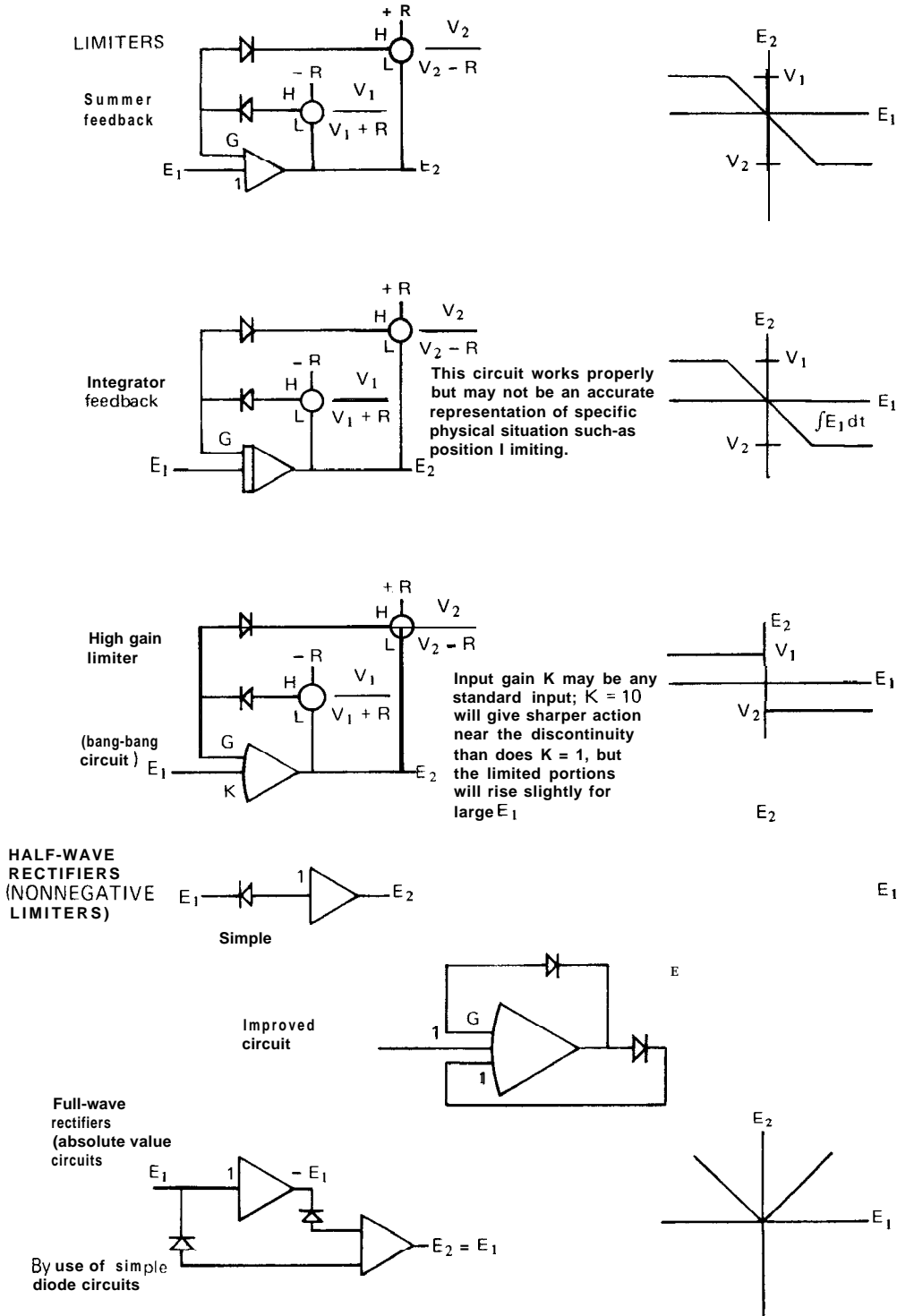
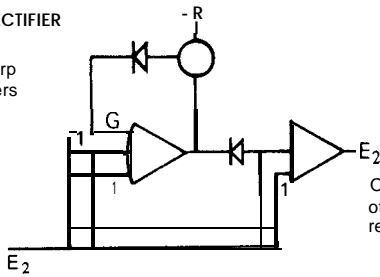


Fig. 44. Common diode circuits.

FULL-WAVE RECTIFIER

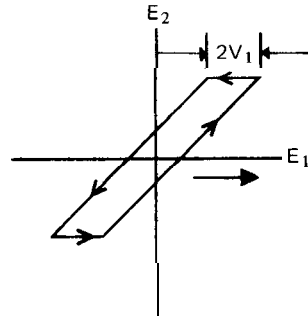
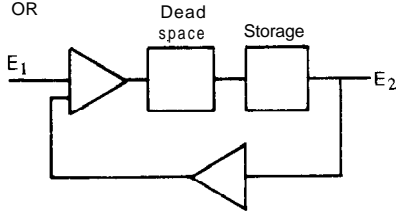
By use of sharp cutoff rectifiers



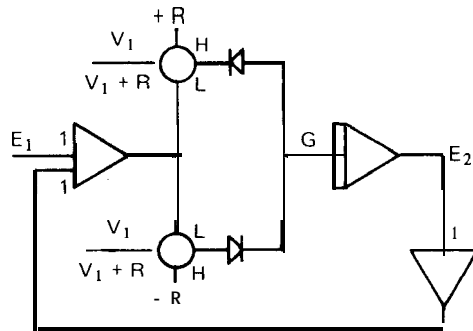
Other circuits make use of various half-wave rectifier connectors

REPRESENTATION OF HYSTERESIS OR BACK LASH

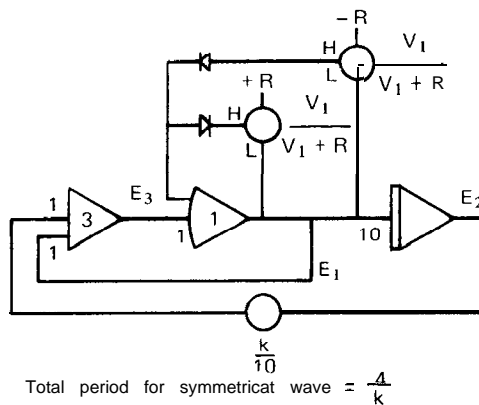
General scheme



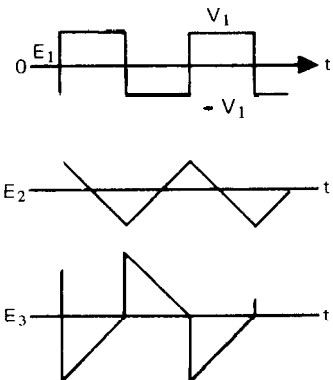
Method using diodes



Triangular and square wave generator



Total period for symmetrical wave = $\frac{4}{k}$



Obtain nonsymmetrical forms if desired by nonsymmetrical limiting or by adding input bias

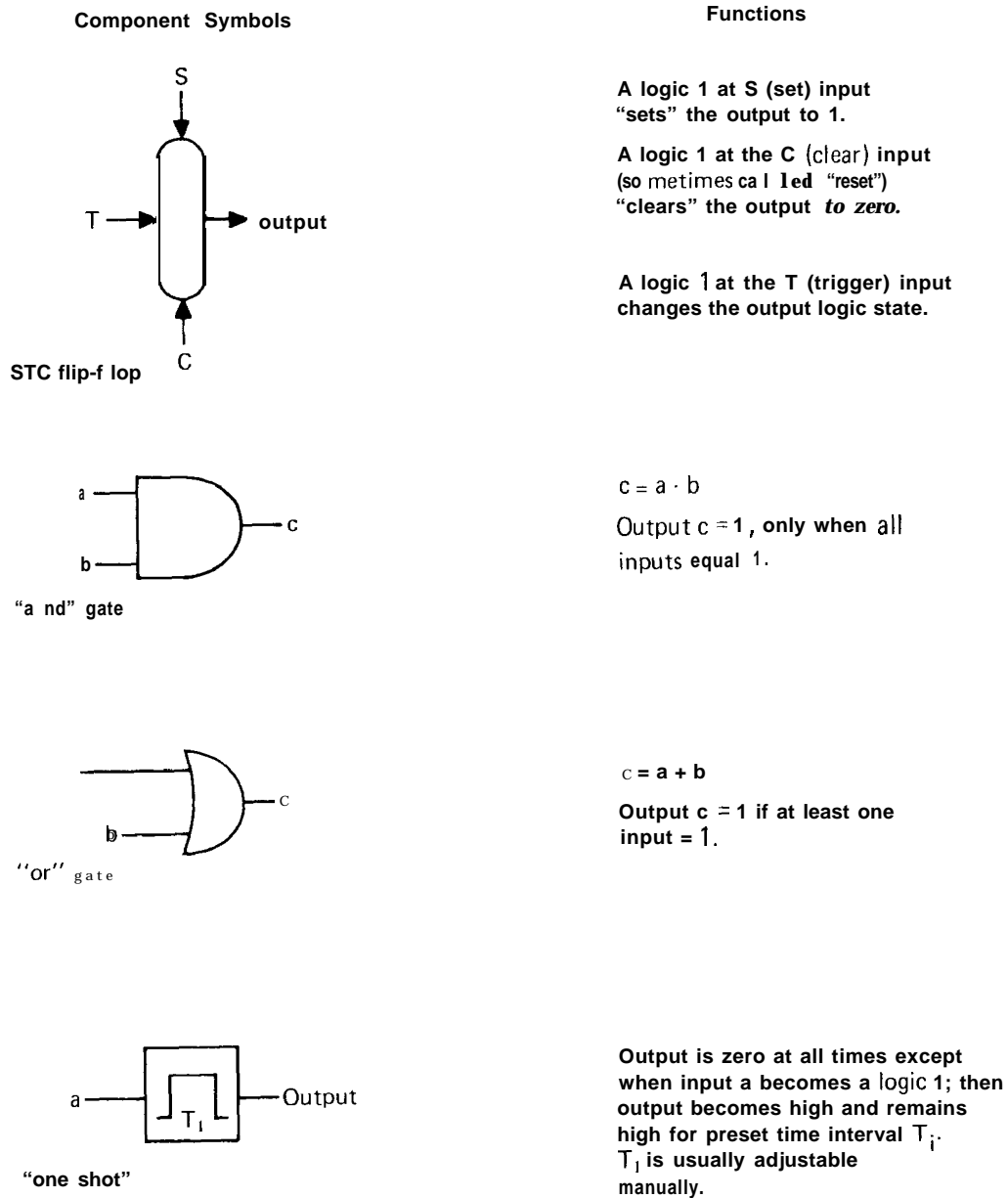


Fig. 47. Symbols and functions of common digital logic modules.

OUTPUT EQUIPMENT

The classical analog output is a multichannel voltage-time recorder. The usual recorders associated with modern analog computers are eight channels wide, write rectangulary, and have adjustable voltage scales and chart speeds. As many variable outputs as desired may be recorded simultaneously, provided one has a sufficient number of recorders. The results produced are called "time histories." The accuracy is good to about 0.25% of the voltage range at which one is recording, and the bandwidth is about 100 Hz.

For wider bandwidth recording, an optical recorder (oscillograph) or some form of magnetic tape recorder must be used.

To obtain X-Y graphs (for example, pressure vs. flow), where any variable Y is plotted as a function of any other variable X (as distinct from plotting X and Y as functions of time), an X-Y "plotter" is used. These may have graph paper from as small as 8 1/2 by 11 in. to as large as 45 by 60 in. The typical X-Y plotter is rather slow, having a useful bandwidth of under 5 Hz. The larger plotters are usually restricted to 1 Hz or slower. Accuracy may be good to about 0.1% of the voltage range at which one is plotting.

Many analog computers are equipped with digital line printers that can print at rates of three to ten lines per second. A "line" consists of the voltage value (converted to four- or five-place decimal digital format) plus sign, plus the address of the analog component being "read" or printed. Hybrid computers can increase this speed considerably by using high speed conversion equipment. With high-speed analog-to-digital conversion directly linked to a digital core memory, one can obtain rates of 100,000 "lines" per second. All digital outputs are usually accurate to four decimal places (0.01% of full scale). Since all outputs of the analog computer are continuous voltages, they may also be used to drive voltmeters, ammeters, oscilloscopes, servos, etc., allowing one an infinite variety of output displays.

PROGRAMMING

Amplitude Scaling. Differential and/or algebraic equations, in order to be mechanized on the analog computer, must first be converted to voltage equations. A scale factor, or volts per physical unit ratio, must be chosen for all the dependent variables. Scale factors are chosen from estimated ranges of the problem variables. These estimates are usually "ed-

ucated guesses," derived from the engineer's experience in his field. If the first estimates prove to be poor, scale factors can be changed at the computer.

Having determined the amplitude scale factors, the problem variables in the mathematical equations are replaced by the voltages or machine units representing them, and adjustments are made to the coefficients throughout the equations in order to maintain equality.

The equations are thus changed into voltage or machine unit equations from which a computer circuit diagram can be drawn.

Time Scaling. With the all-electronic, high-speed analog computers available today, extremely high solution speeds (as short as several milliseconds) as well as very slow solutions (lasting several hours) can be obtained with the same computer. The choice of the solution time is largely dependent on factors external to the computer, such as the method of recording or displaying the solution, the need for tying into real hardware (hence the necessity of operating in "real time"), or the desire to display results to a "man in the loop," etc. A time-scale change is defined by the equation $T = \beta t$, where T is machine time, t is original problem time, and β is the time scale factor and has the units of machine time/original problem time.

In order to slow down a problem (i.e., to cause machine time to be larger than original problem time), β is made greater than unity; to speed up a problem (i.e., to cause machine time to be smaller than original problem time), β is made less than unity.

An objective of time scaling is to change computer time with respect to original problem time, but without causing a change in the original equations, without giving rise to new definitions of derivatives, and without changing any amplitude scaling. For details on how to program an analog computer, see Hausner (1971) or Huskey and G. A. Korn (1962).

MATHEMATICAL APPLICATIONS

It is well known that the analog computer can solve nonlinear, ordinary differential equations, and therefore it is typically used in engineering design and real-time simulation. For up-to-date applications to the various engineering and science disciplines, the reader is referred to the publication *Simulation*. What is not too well known is that the analog computer can be used effectively to solve a

ANALOG TO DIGITAL CONVERTERS

variety of other mathematical equations, and can also be used for analog data analysis. For example, algebraic equations, both linear and nonlinear, are readily solvable. Problems in complex variables are likewise amenable to solution by the analog computer (see Hausner, 1971). These types of problems, while amenable to analog solution, are not of major importance to analog computation. Partial differential equations (PDE) and statistical applications, on the other hand, are of importance in the analog field. For a comprehensive discussion of the solution of the PDE by analog techniques the reader is referred to Hausner (1971), while for statistical applications the reader is referred to the section "Random Process Studies" in Huskey and G. A. Korn (1962).

REFERENCES

1947. Ragazzini, J., R. H. Randall, and F. A. Russell. "Analysis of Problems in Dynamics by Electronic Circuits," *Proc. IRE*, Vol. 35, pp. 444-452.
1955. Rodel, J. In H. M. Paynter (Ed.). *Palimpsest on the Electric Analog Art*, George H. Philbrick Researches, pp. 27-47.
1962. Huskey, H., and G. A. Korn. *Computer Handbook*. New York: McGraw-Hill.
- Simulation*, published since 1963 by Simulation Councils, Inc., La Jolla, Calif. (Describes current analog and hybrid computer work.)
1964. Korn, G. A., and T. M. Korn. *Electronic Analog and Hybrid Computers*. New York: McGraw-Hill.
1971. Hausner, A. *Analog & Hybrid Computer Programming*. Englewood Cliffs, N. J.: Prentice-Hall.
1971. Holst, P. A. "A Note of History," *Simulation*, Vol. 17, No. 3, September, pp. 131-135.

A. I. RUBIN

ANALOG TO DIGITAL CONVERTERS. See DIGITAL TO ANALOG CONVERTERS.

APPLICATIONS OF COMPUTERS.

See ADMINISTRATIVE-BUSINESS APPLICATIONS; ARTS APPLICATIONS; CONTROL APPLICATIONS; CREDIT SYSTEM APPLICATIONS; ECONOMIC AP-

PLICATIONS; ENGINEERING APPLICATIONS; HUMANITIES APPLICATIONS; MEDICAL APPLICATIONS; PLANNING, COMPUTER APPLICATIONS IN; REAL TIME APPLICATIONS; SCIENTIFIC APPLICATIONS; SOCIAL SCIENCE APPLICATIONS; and STATISTICAL APPLICATIONS.

APPLICATIONS PROGRAMMING

For articles on related subjects see PROGRAMMING LANGUAGES; and SYSTEMS PROGRAMMING.

For articles on related terms see ACCESS METHODS; DATA BASE AND DATA BASE MANAGEMENT; SORT-MERGE PACKAGE; and SUBROUTINE.

Applications programs are the programs that are written to solve specific problems, to produce specific reports, to update specific files. The programming languages that are mostly used in applications programming are Fortran for scientific applications and Cobol for data processing applications. Special Report Program Generator (RPG) languages are used on small data processing computers, and languages like Basic and APL are used extensively in time-sharing systems. The language PL/I was introduced by IBM in the hope that it would prove attractive over almost the entire spectrum of applications programming and might eventually supersede both Fortran and Cobol. The resulting language uniformity would have many advantages, but the impact of PL/I has not been very great. Fortran and Cobol remain the standard applications programming languages.

The ultimate aim of all software is to make it possible for the applications programmer to perform his job well and to write programs that produce results and make effective and efficient use of the computing system. Applications programs make use of subroutine libraries and special packages such as sort-merge systems and data access and data management systems. Most well-designed operating systems provide the applications programmer with special tools for analyzing and debugging his programs.

There are very large applications systems such as airline reservations systems and on-line banking and merchandising systems in which many considerations of systems programming and of applications programming are intermixed.

S. ROSEN

APPROXIMATION. See **Chebyshev Approximation**; and **Least Squares Approximation**.

APPROXIMATION THEORY

For articles on related subjects see **Chebyshev Approximation**; **Least Squares Approximation**; and **Numerical Analysis**.

Approximation theory concerns the following problem: Given a function $f(x)$ defined for x in a prescribed set X , a family of functions G , and a metric $d(f, g)$ (a mathematical prescription for measuring the distance between two functions), determine a function $g(x)$ in G which is "close" to $f(x)$ for x in X . For computer applications, $f(x)$ is typically a continuous function of one real variable, X is a real interval, G is a family of polynomials or of rational functions (ratios of polynomials), and the metric is either a least squares metric

$$d_2(f, g, w) = \int_X [f(x) - g(x)]^2 w(x) dx,$$

or the Chebyshev metric

$$d_\infty(f, g, w) = \max_X |[f(x) - g(x)]w(x)|,$$

where $w(x)$ is a weight function. For the Chebyshev metric, the weight is usually either $w(x) = 1$ or $w(x) = 1/f(x)$, where $f(x)$ is assumed not to vanish for x in X . This latter weighting is most useful when $f(x)$ varies considerably in magnitude across the interval X . Basic theorems examine the existence, uniqueness, and characterization of $g(x)$, sometimes in very abstract settings. In this article we will concentrate on the Chebyshev metric, which is more important than the least squares metric in the generation of approximations to functions to be used on a computer.

Let $f(x)$ be defined and continuous over a finite real interval X . The theoretical justification for using the Chebyshev metric (with $w(x) = 1$) is the **Weierstrass approximation theorem**, which asserts the existence of real polynomials that are arbitrarily close to $f(x)$ over the entire interval X . These polynomials are often obtained by appropriately

truncating an infinite power series expansion of the function,

$$f(x) = \sum_{k=0}^{\infty} a_k x^k,$$

provided the series converges to $f(x)$ over X , i.e., provided that for any fixed value of x in X and any $\epsilon > 0$, there is an integer N such that all partial sums

$$s_n(x) = \sum_{k=0}^n a_k x^k, \quad n > N$$

differ from $f(x)$ by less than ϵ . Such expansions are unique whenever they exist.

Some of the more important methods for generating series expansions are based upon the analytic properties of the function. Let $f(x)$ be continuous and have continuous derivatives of all orders at some point x_0 in X . Then the *Taylor series* expansion of $f(x)$ about x_0 is given by

$$f(x) = \sum_{k=0}^{\infty} a_k (x - x_0)^k$$

$$a_k = f^{(k)}(x_0)/k! = \frac{1}{k!} \left. \frac{d^k f(x)}{dx^k} \right|_{x=x_0}.$$

Since this expansion is based upon a detailed knowledge of the function at x_0 , the Taylor polynomials $g_n(x)$ of degree n , obtained by truncating the series, approximate $f(x)$ well for small $|x - x_0|$, but the error $f(x) - g_n(x)$ typically grows monotonically in magnitude with increasing $|x - x_0|$. Frequently $\max |f(x) - g_n(x)|$ occurs at one of the boundaries of X . For example, Fig. 1 shows the error associated with the fourth-degree Taylor polynomial approximation to e^x over $[-1, 1]$, where the Taylor series is

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \dots + \frac{x^n}{n!} + \dots$$

A function $f(x)$ has a pole of finite integer order n at x_0 whenever the Taylor series for $(x - x_0)^n f(x)$ exists for $m = n$, but fails to exist for smaller integer values of m . The *Laurent series* is then given by $(x - x_0)^{-n}$ times the Taylor series for $(x - x_0)^n f(x)$. As an example, the Laurent series for $\csc(x)$, which has an isolated pole of order 1 at $x = 0$, is

$$\csc(x) = \frac{1}{x} + \frac{x}{6} + \frac{7x^3}{360} + \frac{31x^5}{15,120} + \dots,$$

which converges for $|x| < \pi$. When the function

APPROXIMATION THEORY

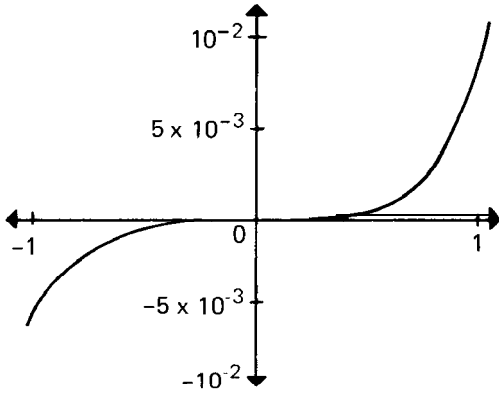


Fig. 1. Error $e^x - g_4(x)$, $g_4(x) = 1 + x + (x^2/2!) + (x^3/3!) + (x^4/4!)$ for approximation over $[-1, 1]$ by fourth degree Taylor polynomial.

has an isolated pole of infinite order, the Laurent series takes the form

$$f(x) = \sum_{k=-\infty}^{\infty} a_k x^k,$$

where the derivation of the coefficients a_k generally involves methods from the theory of functions of a complex variable. The expansion

$$\arctan(x) = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + \dots,$$

valid for $|x| > 1$, is of this type. Truncation of Laurent series leads to rational approximations for $f(x)$.

If the interval X is semi-infinite, $X = [b, \infty)$, $b > 0$, a divergent asymptotic expansion of $f(x)$,

$$f(x) \sim \sum_{k=0}^{\infty} a_k x^{-k},$$

may yield useful rational approximations to $f(x)$ even though it does not converge to $f(x)$ for any finite value of x . Let $s_n(x) = \sum_{k=0}^n a_k x^{-k}$ be the partial sum of the asymptotic series. Then, for any fixed value of x , there is an n which minimizes the error $|f(x) - s_n(x)|$. For fixed n , and x sufficiently large, the error can be made as small as desired. Thus, for a particular $\epsilon > 0$, it is usually possible to choose first an n and then an X (i.e., a, b) so that $s_n(x)$ approximates $f(x)$ to within ϵ over X in the Chebyshev metric. The derivation of an asymptotic series is often a difficult task involving advanced mathematical tools. As with power series, the expansions are unique.

If the Taylor series expansion for $f(x)$ exists, then the *Padé table* for $f(x)$ is the array of rational approximations

$$R_{mn}(x) = \frac{p_0 + p_1 x + \dots + p_m x^m}{1 + q_1 x + \dots + q_n x^n},$$

characterized by the property that the power series expansion of $R_{mn}(x)$ is identical to the Taylor series expansion through terms in x^{m+n} . The entries $R_{mn}(x)$ are the Taylor polynomials, and the entries $R_{01}(x)$, $R_{11}(x)$, \dots along the main diagonal are the successive convergents of a Stieltjes continued fraction, or *S-fraction*, expansion of $f(x)$.

$$f(x) = \frac{a_0}{1 - \frac{a_1 x}{1 - \frac{a_2 x}{\dots}}}$$

These latter elements are often better approximations to $f(x)$ than are the Taylor polynomials of degree $m + n$. All elements of the Padé table agree with $f(x)$ exactly at the point of expansion, but $f(x) - R_{mn}(x)$ tends to grow as x moves away from that point. As an example, the S-fraction expansion of e^x is

$$e^x = \frac{1}{1 - \frac{x}{1 + \frac{x/2}{1 - \frac{x/6}{1 + \frac{x/6}{\dots}}}}}$$

The corresponding Padé approximation $R_{22}(x)$ is obtained by truncating the S-fraction to just the terms given above, and is

$$R_{22}(x) = \frac{12 + 6x + x^2}{12 - 6x + x^2}$$

Fig. 2 shows the error $e^x - R_{22}(x)$ over the interval $[-1, 1]$. Note that the maximum error is less than half of that associated with the fourth-degree Taylor polynomial.

By sacrificing accuracy in the neighborhood of the point of expansion, it is possible to distribute the error over the interval of approximation and to

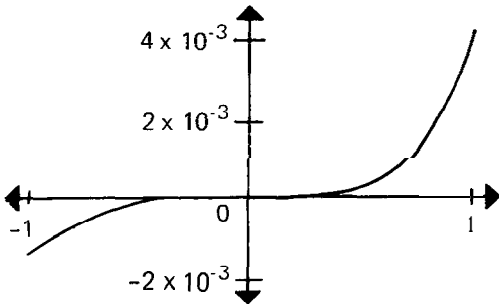


Fig. 2. Error $e^x = R_{n,n}(x)$ for approximation over $[-1, 1]$ by Padé element.

obtain better approximations to $f(x)$ over X in the sense of the Chebyshev metric. The rational Chebyshev, or **minimax**, approximation to $f(x)$ of degree (m, n) is that rational function $R_{mn}^*(x)$, which minimizes $d(f, R_{mn}, w)$. Basic theorems assert that such an $R_{mn}^*(x)$ exists, is unique, and is characterized by the error $[f(x) - R_{mn}^*(x)]w(x)$ achieving its maximum magnitude with alternating sign a prescribed number of times as x moves across the interval X . The determination of $R_{mn}^*(x)$ is not easy, but the characterization theorem leads to algorithms, such as the Remes algorithm, for computing approximations close to $R_{mn}^*(x)$.

The Chebyshev polynomials

$$T_n(x) = 2^{1-n} \cos(n \cos^{-1}x), \quad -1 \leq x \leq 1,$$

are instrumental in the generation of near-minimax polynomial approximations. If $f(x)$ is continuous and sufficiently smooth, then

$$f(x) = \frac{1}{2} a_0 T_0(x) + \sum_{k=1}^{\infty} a_k T_k(x), \quad -1 \leq x \leq 1,$$

where

$$a_k = \frac{2}{\pi} \int_{-1}^1 \frac{f(x) T_k(x)}{(1-x^2)^{1/2}} dx,$$

is the *Chebyshev polynomial expansion* of $f(x)$. (This is related to the *Fourier series* for $f(x)$ by the change of variable $w = \cos^{-1}x$). The partial sums of this expansion are the best polynomial approximations to $f(x)$ for the metric $d_2[f, g, 1/(1-x^2)^{1/2}]$ and are very close to the **minimax** polynomial approximation to $f(x)$ in most cases. As an example, the coefficients in the Chebyshev series expansion for e^x are $a_k = 2I_k(1)$, where the I_k are modified Bessel func-

tions. Truncation of this series after five terms leads to the approximation

$$g(x) = 1.000045 + 0.997308x + 0.499197x^2 + 0.177347x^3 + 0.043794x^4$$

for the interval $[-1, 1]$. The maximum error associated with this approximation is only about one-twentieth of that associated with the fourth-degree Taylor polynomial (see Fig. 3).

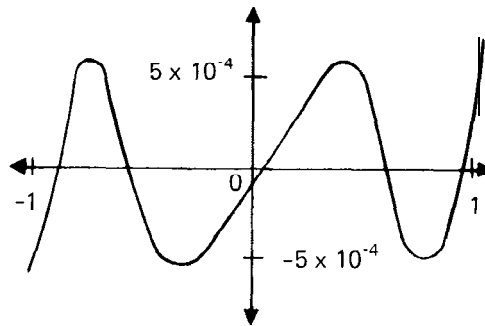


Fig. 3. Error $e^x - \sum_{k=0}^4 a_k T_k(x)$ for approximation over $[-1, 1]$ by truncated Chebyshev series.

Legendre, Jacobi, Hermite, Laguerre, and Gegenbauer polynomials are other important families of polynomials similarly associated with particular choices of weights and intervals in least squares approximation. Since power series expansions are unique, the expansion of $f(x)$ in polynomials from any of these families can be formally obtained by replacing each x^k in the power series by its exact representation in polynomials of the family and then collecting terms. This "rearrangement" of the power series may alter the convergence of the series so that the new series converges for a larger (or smaller) interval than the original series.

Lanczos' telescoping, or economizing, process is similar to this rearrangement process. Starting from a truncated power series, such as a Taylor polynomial, over the interval $[-1, 1]$, the degree of the polynomial is lowered by successively replacing the highest-order term x^n by the polynomial

$$P_{n-1}(x) = x^n - 2^{1-n} T_n(x),$$

which is the **minimax** approximation to x^n by a polynomial of degree less than n . The approximation error introduced at each step tends to distribute the cumulative error over the interval of approximation

ARCHITECTURE, COMPUTER

so that the polynomials in the resulting sequence tend to be better approximations to $f(x)$ than the corresponding truncations of the original power series, but they are not as good as those obtained by truncating the Chebyshev polynomial expansion. For example, the approximation

$$g(x) = 1 + 0.997396x + 0.5x^2 + 0.177083x^3 + 0.041667x^4$$

to e^x is obtained by truncating the Taylor series after six terms and replacing x^5 by $P_4(x) = (20x^3 - 5x)/16$. The corresponding maximum error over $[-1, 1]$ is more than four times that of the corresponding truncated Chebyshev series, but only one-fifth that of the fourth-degree Taylor polynomial (see Fig. 4).

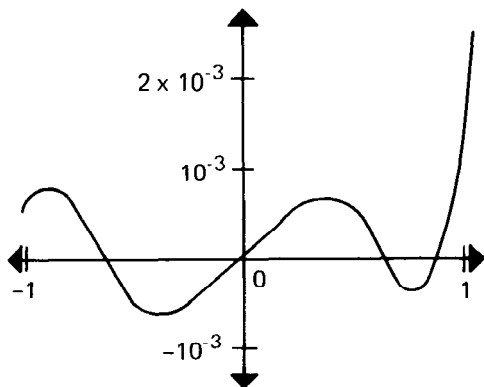


Fig. 4. Error $e^x - g(x)$ for approximation over $[-1, 1]$ by fifth-degree Taylor polynomial telescoped to fourth degree.

An extensive theory of approximation exists for functions of a complex variable and for multivariate functions (functions of two or more real variables). The theory relies heavily upon convergent or asymptotic power series and continued fraction expansions. The Taylor and Laurent series and the Padé table extend to the complex case directly. While the theory of minimax approximation generalizes to these functions, the generalizations are not very useful (e.g., uniqueness is lost in the multivariate case) and reliable algorithms for generating the approximations do not exist. Except for certain elementary functions, direct approximations to complex or multivariate functions are not often used in computer applications. Instead, indirect evaluation methods based upon recurrence relations, differential equations, etc., are used.

REFERENCES

- 1967. Meinardus, G. *Approximation of Functions: Theory and Numerical Methods*, translated by L. Schumaker. New York: Springer-Verlag.
- 1968. Fike, C. T. *Computer Evaluation of Mathematical Functions*. Englewood Cliffs, N.J.: Prentice-Hall.
- 1970. Cheney, E. W. *Introduction to Approximation Theory*. New York: McGraw-Hill.

W. J. CODY

ARCHITECTURE, COMPUTER. See COMPUTER ARCHITECTURE.

ARGUMENT

For articles on related subjects see **DATA TYPE**; **MACROINSTRUCTION**; **PROCEDURE**; **SUBPROGRAM**, **CALLING**; and **SUBROUTINE**.

In strict analogy to mathematics, where an argument of a function is the value of a variable used to evaluate the function, an argument in computing is a value supplied to a procedure, a subroutine, or a macroinstruction which is required in order to evaluate the procedure, subroutine, or macro. Another term used interchangeably with argument is "parameter."

Two different kinds of arguments need to be distinguished: "dummy" or "formal" arguments, and "actual" or "calling" arguments. A dummy argument is an argument used in the definition of a procedure or macro; an actual argument is that which is substituted when the procedure or macro is invoked. For example, Fig. 1 displays a **Fortran SUBROUTINE** subprogram to compute the solution of a quadratic equation.

$$ax^2 + bx + c = 0.$$

The variables **A**, **B**, **C**, **MODE**, **x1**, and **x2** in Fig. 1 are all dummy arguments. If this subroutine were to be used to compute the roots of

$$10.7X^2 + (R1 + 6.23)X + S*S = 0 \quad (1)$$

where **R1** and **s** are variables appearing elsewhere in

```

SUBROUTINE QUAD (A,B,C,MODE,X1, X2)
  DISC = B*B - 4.0*A*C
  IF(DISC.LT.0.) GO TO 3
C PROGRAM IGNORES CASE A=0
C MODE PARAMETER SET TO 0 IF ROOTS ARE
C REAL AND TO 1 IF THEY ARE COMPLEX
  MODE = 0
C TWO REAL ROOTS COMPUTED SO AS TO
c AVOID DIFFERENCE OF TWO NEARLY
c EQUAL QUANTITIES
  IF(B.LE.0) GO TO 5
    X1 = -B / SQRT (DISC)
    GO TO 7
5    X1 = -B + SQRT (DISC)
7    X2 = C/(X1*A)
  RETURN
3    MODE = 1
  X1 = B/(2.0*A)
  X2 = SQRT (- DISC)/(2.0*A)
  RETURN
END

```

Fig. 1. Quadratic equation subprogram.

the program, the statement

```
CALL QUAD (10.7, R1 + 6.23, S*S, J, Y, Z)
```

might be given. Each argument in the `CALL` statement is an actual argument, i.e., the argument that will be associated with the dummy arguments in the subroutine definition. Thus, when `QUAD` is executed in response to `CALL`,

- The values used for `A`, `B`, and `C` will be, respectively, 10.7, `R1 + 6.23`, and `S*S`, with the latter two being evaluated using the current main program values for `R1` and `S`.
- The variable `J` in the main program will be set equal to the value of `MODE` in the subprogram.
- The main program variables `Y` and `Z` will contain the results of the solution of Eq. (1) after execution of `QUAD`.

Formal arguments are always required to be identifiers, but, as the example above indicates, actual arguments may be identifiers or numbers or arithmetic expressions. Most languages allow great generality in the form of the actual arguments, although there may be requirements that the calling arguments have the same “type” or “mode” as the formal arguments (i.e., a real calling argument if the formal argument denotes a real variable).

Subprogram arguments may also be classified as “input” or “output” arguments, with the former denoting arguments provided to the subprogram and the latter the arguments that convey results back to the main program. In the example given in Fig. 1, `A`, `B`, and `C` are input arguments and `MODE`, `X1`, and `X2` are output arguments. Sometimes an argument may be both an input and output argument; for example, when a procedure to compute the next prime number receives as input the variable `P` denoting the current prime number and returns the value of the next prime number to `P`. Sometimes the arguments of a subprogram may be *implicit*; i.e., they are not stated explicitly in the statement heading the subprogram. This happens in block-structured languages when a procedure in a subblock uses variables global to that block. It happens in **Fortran** when arguments are held in so-called **COMMON** storage that is accessible and known by both the main program and the subprogram.

REFERENCE

- 197 I. Ralston, A. *An Introduction to Programming and Computer Science*. New York: McGraw-Hill.

A. RALSTON

ARITHMETIC. See **INTERVAL ARITHMETIC**; and **SIGNIFICANCE ARITHMETIC**.

ARITHMETIC, COMPUTER

For articles on related subjects see **COMPLEMENT**; **NUMBERS AND NUMBER SYSTEMS**; **PRECISION**; **ROUND OFF ERROR**; **SIGNIFICANCE ARITHMETIC**; and **SIGNIFICANT DIGIT**.

For article on related term see **WORD LENGTH**, **VARIABLE**.

The earliest electronic computers were developed in the late 1940s to fill a need for fast arithmetic engines that could solve a variety of problems, many of them military. Although computers, as general symbol manipulators, now solve many problems that do not involve arithmetic computation, numerical calculations are still of vital

ARITHMETIC, COMPUTER

importance in computer applications. Therefore, how computers perform arithmetic and, in particular, how computer arithmetic differs from ordinary hand computation are topics that should be understood by anyone who uses computers.

Storage of Numbers in Computers. In most computers, a single number occupies a single memory unit (in word-oriented memories) or a fixed number of memory units (in *character* or byte-oriented memories; e.g., four bytes in IBM 360-370 series computers), which are usually called a "word" or "full word." Such numbers are sometimes called "single precision" numbers in contrast to double precision numbers, which are used when additional accuracy is required and which occupy twice the memory space of single precision numbers (e.g., eight bytes in IBM 360-370 computers). (Years ago, some computers-epitomized by the IBM 1400 series and the IBM 1620—had variable word lengths and therefore could handle variable length numbers. Few such computers are still in use.)

A given number may be stored in one of two modes: *fixed point* or *floating point*. Fig. 1(a) illustrates the storage of .15625 as a fixed-point number in a word-oriented computer memory with 36 bits per word. (Throughout this article, numbers in the text will be decimal numbers and those illustrated in computer storage will be binary.) Two points are worth noting about this example:

1. The left-hand bit (S) in Fig. 1 represents the sign of the number, 0 for + and 1 for -. (Hereafter in this article we will, for convenience, use only positive numbers in examples. Storage of negative numbers can be either in absolute value and sign form or in complement form.)

2. The binary point (i.e., the "decimal" or *radix* point) is assumed to be at the left end of the number; hence the name "fixed-point" numbers. All computers must embody an assumption on the invariant

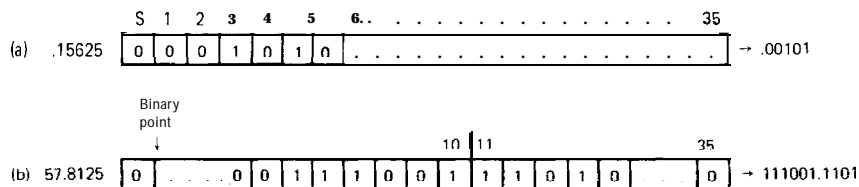
location of the binary point in fixed-point numbers. In most computers this assumption (used throughout this article) is the one shown in Fig. 1. However, some computers (e.g., CDC 6000 series) assume the binary point to be at the right-hand end of the number.

Hardware for storing and manipulating numbers in fixed-point form was the only kind available in early computers, but handling them and doing arithmetic with them created major problems:

1. How could numbers with magnitudes of 1 or greater be handled? The answer to this is that such numbers can be stored as fixed-point numbers but the programmer must be aware of and keep track of the implicit location of the radix point as such numbers are used in computations. Thus, 57.8125 could be stored as shown in Fig. 1(b).

2. More significant are the problems with numbers whose implicit radix points are in different positions. How could these be added, subtracted, multiplied, and divided? As a simple example, consider adding the numbers in Figs. 1(a) and 1(b). Before this can be done by adding the contents of the words, bit by bit, one of the numbers must be shifted relative to the other. The process of such *shifting* and the associated problem of choosing the location of the implicit binary point for numbers before they are stored is called "scaling." The difficulty and tediousness of scaling for all but the simplest computations led to the introduction of floating-point hardware capabilities, which are now used for the vast majority of all numerical computation on computers.

Floating-point numbers are more than the solution to the above problems because they also allow computations where the range of the magnitude of the numbers is very large, larger than can be handled with fixed-point numbers except with great diffi-



culty. Floating-point representation of numbers corresponds very closely to what is usually called "scientific notation"; that is, each number is represented as the product of a normal number with a radix point and an integral power of the radix. Thus, for example, the number of Fig. 1(b) might be expressed in scientific notation as

$$578125 \times 10^2$$

with the 578125 being called the "fractional" part and the 2 the "exponent" part; or, borrowing from logarithmic terminology, the mantissa and characteristic, respectively. This notation is called "floating point" in computer arithmetic because the radix point of the entire number (57.8125 in Part (b) of Fig. 1) is not fixed but can "float," depending upon the value of the exponent.

In order to store such numbers in memory units of the same length as single precision fixed-point numbers, separate portions of each unit must be assigned to the fractional part and the exponent part. Standard ways of doing this in computers having a 36-bit word memory and four 8-bit bytes to a word (like the IBM 360-370 computers) are shown in Figs. 2(a) and 2(b) for the number 578125×10^2 . The following comments pertain to these figures.

1. A notation sometimes used for the number in Fig. 2(a) is (8,27), indicating 8 bits for the exponent part and 27 for the fractional part. Similarly, the number in Fig. 2(b) is a (7,24) number.

2. As with fixed-point numbers, the binary point of the fractional part is always assumed to be in the same place, usually at the left end, as in Fig. 2, but sometimes at the right-hand end (in which case the "fractional part" is an integer). The sign bit S always represents the sign of the fractional part.

3. Exponents may be positive or negative, but since the sign bit denotes the sign of the fractional part, the sign of the exponent must be handled by a special mechanism. One possibility would be to let bit 1 denote the sign of the exponent, but a much more common technique is to use "excess-n" nota-

tion. In Fig. 2(a) the 8 exponent bits can represent binary integers from 0 to $2^8 - 1 = 255$. If the desired exponent is X and the exponent part stored in bits 1 to 8 is Y, then $Y = X + 128$ is said to be an excess-128 exponent. Thus, true exponents X from -128 to +127 can be represented by values of Y from 0 to 255. Therefore it is necessary only for the arithmetic unit of the computer to interpret correctly the excess-128 exponent.

4. In IBM 360-370 computers the exponent is considered to be an excess-64 binary number, which is interpreted as a power of 16 because many of the internal operations of this series of computers are hexadecimally oriented. Thus, the exponent shown in Fig. 2(b) is interpreted as 16^1 .

5. The range of exponents in the (8,27) case is from 2^{-128} to 2^{+127} , or from about 10^{-38} to 10^{+38} . In the (7,24) hexadecimal case, the range is from 16^{-64} to 16^{63} , or from about 10^{-77} to 10^{+76} . On CDC 6000 series computers, which use an (11,48) binary format with the mantissa radix point at the right, the range is from about 10^{-294} to 10^{322} . Even so, this range is not always adequate, as pointed out in the discussion on overflow in the next section.

The floating-point number in Fig. 2(a) is said to be *normalized* because the most significant bit in its fractional part (bit 9) is *nonzero*. Similarly, the floating-point number in Fig. 2(b) is normalized in a hexadecimal sense because the first hexadecimal digit in the fractional part (0011 = hexadecimal 3) is *nonzero*. Most computers automatically create normalized numbers as the result of floating-point arithmetic operations because in this way the maximum number of significant bits are retained. Some computers allow the programmer to choose whether the result should be normalized or unnormalized. One school of thought believes that leaving results unnormalized gives a better picture of the true accuracy of the retained result.

Double precision floating-point numbers occupy two words, or twice the number of bytes, as single precision numbers. Two formats for these numbers

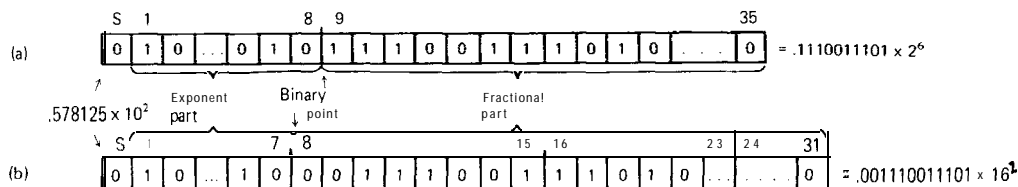


Fig. 2. Floating-point numbers.

ARITHMETIC, COMPUTER

are in use, with the first format below **being** more common today:



where the second fractional part is just an extension of the first and



where the second fractional part is an extension of the first, but where **EXP2** is less than **EXP1** by the number of bits in each fractional part and both halves have the same sign.

Arithmetic, Fixed-Point. The actual mechanics of fixed-point arithmetic are essentially those of ordinary binary arithmetic, given the restriction that negative numbers are generally stored and manipulated in some complement form. However, some aspects of fixed-point arithmetic on computers need to be considered explicitly. In the following examples we assume that fixed-point numbers are binary fractions of magnitude less than 1 (i.e., binary point at the left).

The only snare for the unwary in fixed-point addition and subtraction is the phenomenon known as “overflow.” Since not only the two operands but also the result in addition and subtraction must be less than 1 in magnitude, a result greater than this will not be handled correctly, as illustrated in Fig. 3. Overflow occurs when bit 1 has a carry-out. In some computers this carry-out is discarded; in others, as shown in Fig. 3, it replaces the sign bit, resulting in the example shown in a spurious negative number. In any case, the result is incorrect; normally this is indicated by setting an internal switch which the programmer can test, using a *branch on overflow* instruction.

Overflow cannot occur in fixed-point multiplication, since the product of two factors less than 1 in magnitude is also less than 1. But the multiplication of two n-bit factors results in a 2-n bit product, which cannot be accommodated in an n-bit accumulator

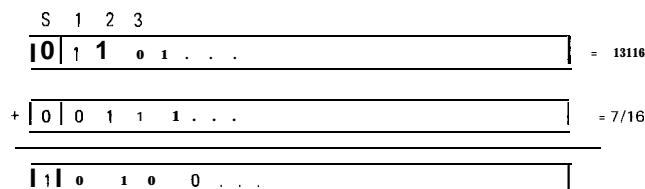


Fig. 3. Overflow in fixed-point numbers.

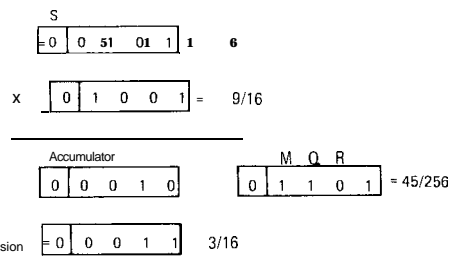


Fig. 4. Fixed-point multiplication.

register in the arithmetic unit. Normally, the least significant n bits are placed in a second register, called the “multiplier-quotient (or MQ) register,” as shown in Fig. 4, in which for convenience we have assumed the word length to be only five bits. If the programmer wishes to retain the full $2n$ -bit product, he may do so, but usually only the most significant n bits are retained after rounding, as shown in Fig. 4.

As with addition and subtraction, division can result in overflow if the dividend has magnitude greater than the divisor. Such an overflow, often called a “divide check,” is ordinarily tested by the programmer with an instruction separate from that used to test additive or subtractive overflow.

The dividend in fixed-point division can usually be double length (or precision), occupying both the accumulator and MQ registers. The single precision quotient is commonly placed in the MQ register and the remainder is placed in the accumulator, as illustrated in Fig. 5.

In actual practice the great majority of fixed-point arithmetic operations are normally performed on integer quantities in higher-level language programs. Appropriate adjustments must be made when the computer assumes that the radix point is at the left-hand end of the number. However, the higher-level language programmer need not be concerned about this, since the higher-level language processor performs all the necessary manipulations. Fig. 6 illustrates what would actually happen in our hy-

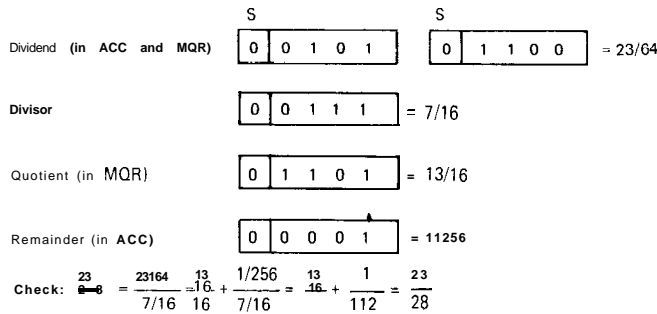


Fig. 5. Fixed-point division.

Dividend
14 made least significant half of double-length dividend to avoid overflow and give correct result directly.

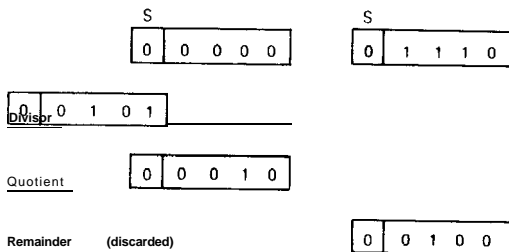


Fig. 6. Integer division in Fortran.

pothetical four-bit plus-sign computer in executing the Fortran statement

$$I = 14/5$$

to give the result 2, since remainders are discarded in Fortran integer division.

Arithmetic, Floating Point. The most difficult operations in floating-point arithmetic are addition and subtraction. A (simplified) algorithm for floating-point addition is the following: Let the two numbers to be added be A and B, and let C be the result. Let the exponent and fractional parts be denoted by E_a , E_b and E_c and F_a , F_b and F_c , respectively.

STEP 1. Set E_c = the larger of E_a and E_b . Assume in what follows that $E_a \geq E_b$.

STEP 2. shift right. Shift F_b to the right $E_a - E_b$ places (which has the effect of giving F_a and F_b the same exponent).

STEP 3. Add. Set $F_c = F_a + F_b$

STEP 4. Normalize. Shift F_c to make 1 its most significant digit and adjust E_c accordingly. This algorithm is illustrated in Fig. 7, assuming a hypothetical computer with a four-bit excess-8 exponent and a six-bit fractional part.

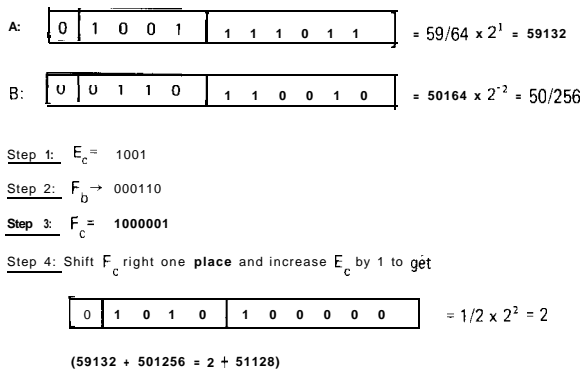


Fig. 7. Floating-point addition.

ARITHMETIC, COMPUTER

In proceeding through these four steps, several points should be considered:

1. Step 3 may result in a kind of overflow in that the two six-bit operands may produce a seven-bit result. But no error results because the "overflow" bit is always retained and shifted right in Step 4.

2. The final computer result is in error by $5/128$ with respect to the true result. Some error is inevitable in general because the right shift in Step 2 may cause the loss of some significant bits of F_b . But if the final fractional part were 100001 instead of 100000, C would equal $2 + 1/16$, in which case the error would be $3/128$ instead of $5/128$. Many floating-point add algorithms would, in fact, achieve this result by *rounding* F_c after Step 3 (by adding 1 in the seventh position) before performing Step 4.

3. Many computers have machine instruction sets that allow the machine or assembly language programmer to choose either normalized or *unnormalized* floating-point operations (i.e., with or without Step 4, except that when a right shift is required, it is always performed). The higher-level language programmer has no such options. Normalized floating-point operations are almost always used in higher-level language processors.

4. Floating-point subtraction or floating-point addition of numbers with different signs may result in a normalization that requires a left shift of F_c . In some computers the arithmetic unit will have retained the bits shifted right in Step 2, and these will now be shifted left to avoid the loss of precision that would be caused if zeros were inserted at the right. Other computers (e.g., the IBM 360-370 series) retain a single *guard digit* (a hexadecimal, not a binary digit, in IBM 360-370) on the right, which can be shifted left during normalization.

Overflow can occur in all floating-point operations when the magnitude of the result exceeds the capacity of the floating-point number system. Although, as noted previously, some computers can accommodate numbers as large as 10^{300} or more, overflow—particularly as the result of a programming *error*—is by no means unheard of. For our hypothetical floating-point number system, an example of overflow as a result of addition is shown in Fig. 8. The result of floating-point overflow is handled either by making the result the largest number possible ($0\ 11\ 11\ 111111$) and setting an indicator that the programmer can test, or by stopping the computation with an appropriate error message.

Underflow, which results from the attempt to produce a *nonzero* result too small (in magnitude) to be accommodated, can also result from any floating-point operation. The usual result is to generate a zero result; sometimes an indicator is also set, which the programmer can test. An example is given in Fig. 8.

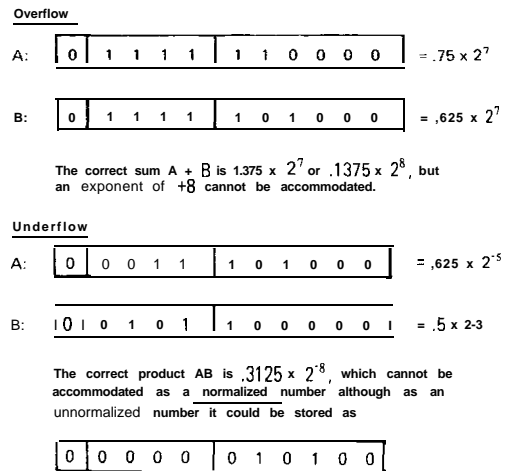


Fig. 8. Floating-point overflow and underflow.

Floating-point multiplication and division only require performing the appropriate action on the fractional parts, rounding the results, adding (for multiplication) or subtracting (for division) the exponents, and then normalizing if necessary. Examples are shown in Fig. 9.

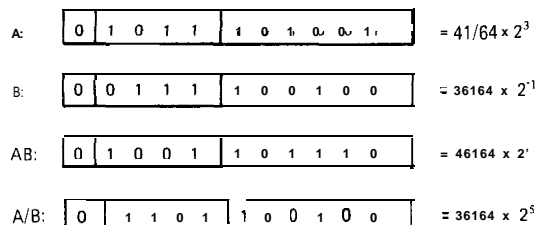


Fig. 9. Floating-point multiplication and division.

Whereas computers do not normally provide hardware for double precision fixed-point arithmetic (which may, however, be programmed), hardware is often provided for double precision floating-point arithmetic. The details, however, do not differ suffi-

ciently from those of single precision arithmetic to merit inclusion here. A comprehensive discussion of floating-point arithmetic is found in Sterbenz (1974).

Computer Arithmetic and Real Arithmetic. Two groups of common laws of arithmetic on the real-number system are:

1. *Associative Laws*

Addition: $a + (b + c) = (a + b) + c$

Multiplication: $a \cdot (b \cdot c) = (a \cdot b) \cdot c$

2. *Distributive Law:*

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

The truth of these laws depends in part on the denseness of the real numbers on the number line or, equivalently, on the ability of numbers to have arbitrarily large numbers of digits. Due to the finiteness of computer arithmetic, these laws are not generally satisfied on computers, although the commutative laws for addition,

$$a + b = b + a$$

and for multiplication,

$$a \cdot b = b \cdot a$$

are generally satisfied. The failure of the associative and distributive laws to be satisfied affects relatively few computations, but on occasion such failure can be crucial. For a further discussion of this see Ralston (1971).

REFERENCES

1963. Flores, I. *The Logic of Computer Arithmetic*. Englewood Cliffs, N.J. : Prentice-Hall.
1969. Knuth, D. E. "Seminumerical Algorithms," in *The Art of Computer Programming*, vol. 2. Reading, Mass.: Addison-Wesley. (This book is the reference on the algorithms by which computer arithmetic is performed.)
1971. Ralston, A. *Introduction to Programming and Computer Science*. New York: McGraw-Hill.
1974. Sterbenz, P. H. *Floating-Point Computation*. Englewood Cliffs, N. J. : Prentice-Hall.

A. RALSTON

ARITHMETIC-LOGIC UNIT

For articles on related subjects see **ADDER**; **ARITHMETIC**, **COMPUTER**; **BOOLEAN ALGEBRA**; **CODES**; **MACHINE INSTRUCTION SET**; **NUMBERS AND NUMBER SYSTEMS**; **OPERAND**; **REGISTER**; and **SHIFTING**.

For articles on related terms see **ALGORITHM**; **MULTIPROCESSING**; **STACK**; and **WORD LENGTH, VARIABLE**.

The ALU is a functional part of the digital computer which carries out arithmetic and logic operations on machine words that represent the operands. It is usually considered to be a part of the central processing unit (CPU). In some computer systems, separate units exist for arithmetic operations (the arithmetic unit, AU) and for logic operations (the logic unit, LU).

Many computers contain more than one AU. For example, a separate Index AU is frequently employed to perform addition or subtraction operations on address parts of instructions for the purpose of indexing, boundary tests for memory protection, etc. Large computer systems employ separate AUs for different classes of algorithms; for example, the IBM System 360, Model 91, contains a fixed-point AU and a floating-point AU. Multiprocessor systems contain several identical ALUs; for example, the ILLIAC IV contains 64 identical ALUs with associated memory modules.

A complete discussion of an ALU must describe its three fundamental attributes:

1. Operands and results.
2. Functional organization.
3. Algorithms.

Operands and Results. Two kinds of ALU organizations can be distinguished with respect to the length of machine words. In machines with **fixed** word length, all words consist of the same number of bits. In machines with variable word length, one byte is the shortest machine word; a typical length of one byte is eight bits. Longer machine words consist of some integral number of bytes.

The operands and results of the ALU are machine words of two kinds: *arithmetic words*, which represent numerical values in digital form, and *logic words*, which represent arbitrary sets of digitally encoded symbols.

Arithmetic words consist of strings of digits. Conventional radix *r* number representations allow

ARITHMETIC-LOGIC UNIT

r values for one digit: 0, 1, . . . , $r - 1$. Practical design considerations have limited the choice of radices to the values 2, 4, 8, 10, and 16. The value of every digit is represented by a set of bits. Radices 2, 4, 8, and 16 employ binary numbers having length of 1, 2, 3, and 4 bits, respectively, to represent the values of one digit. Radix-10 digit values are usually represented by four or five bits. Most commonly used are the four-bit BCD (binary coded decimal) and excess-3, and the five-bit biquinary encodings.

Two methods have been employed to represent negative numbers. In the sign-and-magnitude form, a separate *sign bit* is attached to the string of digits to represent the + and - signs. (Usually 0 represents the +, and 1 represents the - sign.) In the true-and-complement form, the negative value $-x$ is represented as the complement with respect to A of the value x ; i.e.,

$-x$ is represented by $A - x$

Two values of A are used in ALUs: $A = r^{n+1}$ and $A = r^{n+1} - 1$, when x is represented by n digits in the sign-and-magnitude form. An illustration for radix 10 and radix 2 and $n = 4$ is given below.

Sign and Magnitude	$A = 10^5 - 1$ (9s complement)	$A = 10^5$ (10s complement)
+ 4902	04902	04902
- 4902	95097	95098

	$A = 2^5 - 1$ (1 s complement)	$A = 2^5$ (2s complement)
+ 1010	01010	01010
- 1010	10101	10110

The use of complements to represent negative values makes it possible to replace the subtraction algorithm in an ALU by a complementation followed by an addition modulo A .

Other important properties of operands and results are (Aviiienis, 1972) :

- 1. Location of the radix point.
- 2. Use of multiple-precision representations.
- 3. Use of floating-point forms.
- 4. Explicit designation of the number of significant digits in a representation.
- 5. Encoding in error-detecting (or error-correcting) codes.

The use of nonconventional number representations in computers as a means to increase the speed of arithmetic has been proposed. Extensive studies have been made of *residue* number systems (Svoboda, 1962) and of *signed-digit* number systems (Aviiienis and Tung, 1970); however, they have not yet reached practical application in ALU design.

Logic words that serve as operands represent alphanumeric information and are subject only to logic algorithms that are applied to individual bits of the operands. These algorithms are (1) negation for one operand, and (2) the 16 two-variable logic operations for corresponding bits of two operands.

Functional Organization and Algorithms of an ALU. An ALU consists of three types of functional parts: storage registers, operation circuits, and sequencing circuits, as shown in Fig. 1. The inputs and outputs of the ALU are connected to other functional units of the computer, such as the main memory, the program execution control unit, and input/output devices. A bus is most frequently used as the means of connection. In some cases the ALU may be connected to two or more busses within the computer system.

The input information received by the ALU consists of operands, operation codes, and format codes. The operands are machine words that represent numeric or alphanumeric information in the

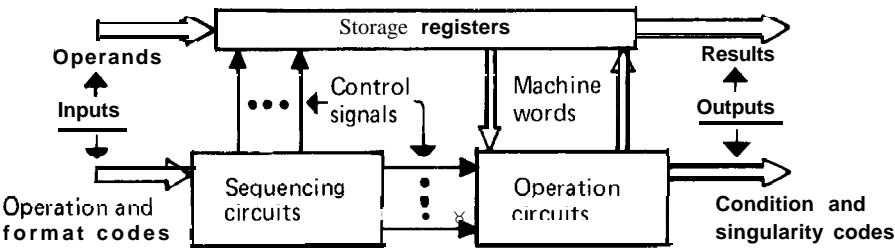


Fig. 1. Functions of an ALU.

form of a string of binary digits (bits). The operation code identifies one operation from the set of available arithmetic and logic operations, and also designates the location (within local storage) of the operands and of the results. The designation of operands is omitted in ALUs with limited local storage; for example, an `ADD` operation code in a single-accumulator ALU always means the addition of the incoming operand to the operand in the accumulator register and storage of the sum in the accumulator. The format code is used when the ALU can operate on more than one type of operand; for example, the `ADD` operation can be specified either for fixed-point or for floating-point operands. Often the operation code and the format code are represented by a single set of bits.

The output information delivered by the ALU consists of results, condition codes, and singularity codes. The results are machine words generated by the specified operations and stored in the local storage registers. The condition codes are bits or sets of bits that identify specific conditions associated with a result, such as that the value of the result is positive, negative, zero; that the result consists of all

zeros, all ones, etc. The singularity codes indicate that the specified operation does not yield a representable result. Examples of singularities are: "overflow," i.e., the value of the result exceeds the allowed range; attempted division by zero; excessive loss of precision in floating-point operations; error caused by a logic fault, etc. Singularity codes usually set a flip-flop in the machine status word,

Internally, the ALU is composed of storage registers, logic circuits that perform arithmetic and logic algorithms, and logic circuits that control the sequence of gating operations within the ALU. The diagram of a simple ALU is shown in Fig. 2.

The ALU contains three registers: the operand register, `OPR`; the accumulator register, `ACC`; and the multiplier-quotient register, `MQR`. Each register contains one machine word, i.e., for a machine word length of n bits, the register consists of n flip-flops.

The gating of words into the ALU registers and from the registers into the operation circuits or out of the ALU is controlled by the sequencing logic (`SL`), which applies a sequence of gate-enabling signals to the gates G_i of Fig. 2. Each sequence corresponds to one of the algorithms provided within

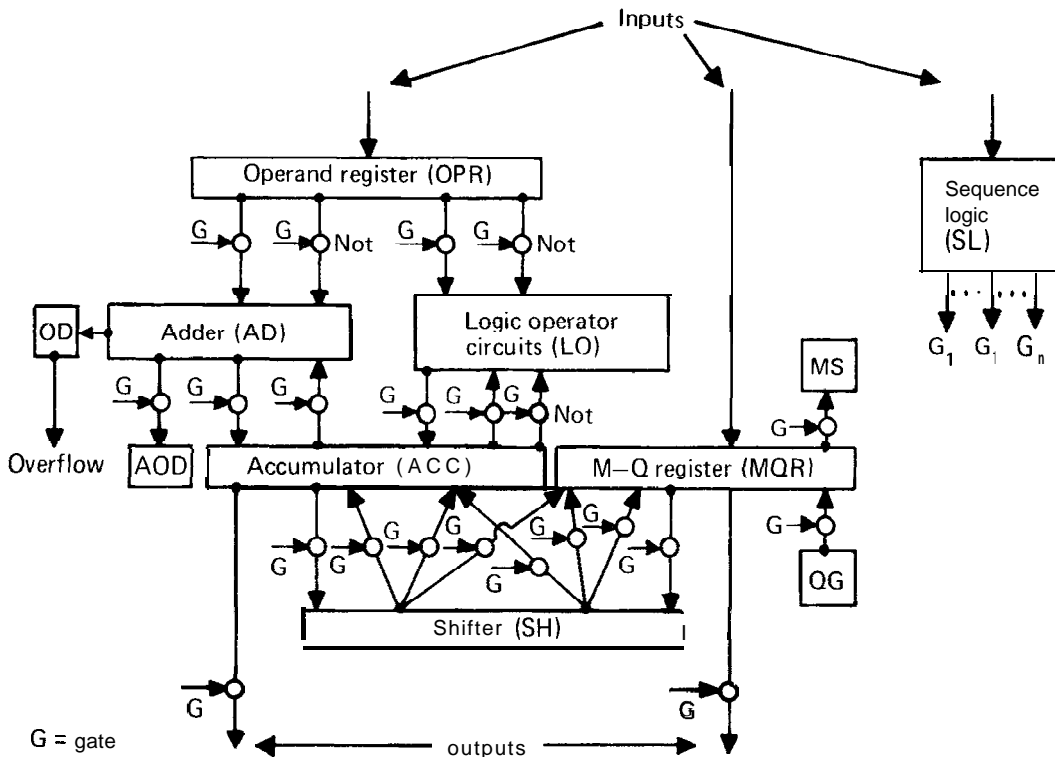


Fig. 2. Organization of an ALU.

ARITHMETIC-LOGIC UNIT

the ALU. The sequencing logic is implemented either in "hard wired" form, using counters and decoding circuits, or by means of a microprogrammed control unit. The sequence of gating signals is initiated by the receipt of the operation and format codes in the ALU.

The operation circuits consist of the adder (AD), the shifter (SH), and the logic operator circuits (LO). The adder forms the sum of the numbers in OPR and ACC and returns it to the ACC. When the length of the sum exceeds the standard word length, the overflow detection (OD) circuit issues an overflow singularity code, and the excess digit of the sum is placed into an overflow digit position (AOD) which is located at the left end of the ACC. Subtraction is usually implemented as complementation of the subtrahend in OPR, followed by its addition to the minuend in ACC. A subtractor may be used instead of an adder in Fig. 2; in this case, subtraction is carried out directly, and addition is implemented as the complementation of the addend in OPR followed by its subtraction from the augend in ACC.

The SH circuits perform left-shift and right-shift operations on the words in ACC and MQR. A single-shift operation displaces every digit in the register to the adjacent position on the left or on the right. Shifts are specified either for one register or for both registers simultaneously, with the rightmost position of ACC adjoining the leftmost position of MQR. There are two classes of shifts:

1. *Circular (or Logic) Shifts.* The rightmost and the leftmost positions of a register are treated as adjacent during the shift.
2. *Arithmetic Shifts.* Digits are discarded from end positions; e.g., during a single right shift, the rightmost digit is lost and the leftmost position is filled in with a specified digit value. The purpose of an arithmetic shift is to multiply (left) or to divide (right) the operand by the radix r .

The shifter is frequently designed as an integral part of the ACC and MQ registers; they are then called "shift registers."

Multiplication and division operations are carried out as a sequence of additions or subtractions and arithmetic shifts. The MQ register serves as the third register for these operations. In multiplication, the multiplicand x is placed into OPR register, the multiplier y into MQ register of Fig. 2, while ACC is cleared to zero. The least significant digit y_0 of the multiplier is sensed by the multiplier sensing (MS) circuit, and x is added y_0 times to the contents of

ACC. Then ACC and MQ registers are arithmetically shifted one position to the right, and the next multiplier digit, y_1 , is sensed by the MS circuit. After all n digits of y have been sensed, the double-length product xy is located in ACC and MQ registers. A round-off operation is needed to reduce the product to single-word length.

To perform division, the dividend is placed into ACC. If the dividend is of double length, MQ register receives its less significant half. The divisor is placed into OP register, and division is carried out as a sequence of trial subtractions and left arithmetic shifts. Quotient digits are generated one at a time in the quotient generation (QG) circuit and inserted at the right end of the MQ register after each shift, beginning with the most significant quotient digit, q_{n-1} . After n steps, the quotient is located in MQ register and the remainder in the ACC register.

The logic operator LO circuits perform the specified logic operation on pairs of bits in corresponding storage positions a_i of ACC and x_i of OP registers. The bits of the result are returned to ACC. The usual set of operations includes NOT (one bit: \bar{a}_i or \bar{x}_i), AND ($a_i \wedge x_i$), OR ($a_i \vee x_i$), EXCLUSIVE-OR ($a_i \oplus x_i$), EQUIVALENCE ($a_i \equiv x_i$), NAND ($\bar{a}_i \vee \bar{x}_i$) and NOR ($\bar{a}_i \wedge \bar{x}_i$); sometimes all 16 two-variable logic operations are provided.

An ALU may be serial, byte-serial, or parallel, depending on how many digits are processed simultaneously in the adder (or logic operator) circuits of Fig. 2. In the serial ALU, the adder adds one pair of digits at once; in the byte-serial ALU, it adds a pair of bytes (consisting of two or more digits); in the parallel ALU, it adds two full machine words. Machines with variable word length have byte-serial ALUs, since the words consist of a varying number of bytes. The time required to complete one addition in the adder circuits is a basic time unit of ALU operation.

The speed of execution of the algorithms in a parallel ALU may be increased by the use of various techniques (Garner, 1965). Addition speed is increased by use of carry-completion sensing, carry-lookahead, or conditional-sum adders. Multiplication is accelerated by multiplier recoding and by the use of multiple-operand carry-save adders. Division employs redundant quotient recoding techniques with approximate estimates, or quadratic convergence which utilizes fast multiplication to generate the quotient (Anderson et al., 1967). The technique of "pipelining" also has been employed to increase the effective throughput of an ALU (Anderson et al., 1967).

The use of more storage registers within the ALU increases the speed of computing by reducing the number of memory accesses. Therefore, 8, 16, or more ALU registers are often used instead of the three registers shown in Fig. 2; each register may perform the function of ACC, OP, or MQ registers. Several ALU registers may be used to hold a *stack* of ALU operands and results.

Some ALUs provide a more extensive set of algorithms, including square root, complex arithmetic, trigonometric functions, etc. Such ALUs are most of ten found in special-purpose computers.

REFERENCES

1962. Svoboda, A. "The Numerical System of Residual Classes," in *Digital Information Processors*, New York: Interscience, pp. 543-574.
1965. Garner, H. L. "Number Systems and Arithmetic," in F. L. Alt (Ed.), *Advances in Computers*, vol. 6. New York Academic Press, pp. 131-194.
1967. Anderson, S. F., J. G. Earle, R. E. Gold-Schmidt, D. M. Powers. "The IBM System/360 Model 9 1: Floating-Point Execution Unit," *IBM Journal of Research and Development*, vol. 11, No. 1 (January), pp. 34-53.
1970. Avižienis A., and C. Tung. "A Universal Arithmetic Building Element (ABE) and Design Methods for Arithmetic Processors," *IEEE Trans. Comput.*, vol. C-19, No. 8 (August) pp. 733-745.
1972. Aviiienis, A. "Digital Computer Arithmetic: A Unified Algorithmic Specification," in *Computers and Automata*. Brooklyn, N. Y.: Polytechnic Institute, pp. 509-525.

A. AVIŽIENIS

ARITHMETIC SCAN

For articles on related subjects see **LANGUAGE PROCESSORS; POLISH NOTATION; and PRECEDENCE.**

In the process of compilation into machine executable code of a program written in a higher-level language, the procedure for examining arithmetic expressions and determining the order of

execution of the operators is often referred to as the "arithmetic scan." Since arithmetic expressions are well formed in that they possess regular properties related to the operands and the operators, many specialized parsing or scanning techniques have been developed. One possible, but impractical, technique is to require the programmer to write arithmetic expressions in fully parenthesized notation (i.e., parentheses must be placed around each pair of operands and its associated operator) so as to obviate the need for knowledge about the relationships between operators in determining the order in which the operations are to be performed.

Most commonly used are transformational systems, which convert the normal infix form (i.e., the form in which the operator is placed between its operands) to a Polish form in which there exists no parentheses and the order of execution of the operators is specified by their positioning. Such a system is needed because of the difficulty of associating operands with operators in infix notation. As an example, consider the Fortran expression

$$(A * X + B) / (C * X + D) \quad (1)$$

which, because of the *precedence* relations among Fortran arithmetic operators, is to be interpreted in fully parenthesized notation, as

$$(((A * X) + B) / ((C * X) + D)) \quad (2)$$

By use of an algorithm (see Ralston, 197 1), which scans across the string in expression (1) from left to right just once, this string can be converted to the Polish postfix string

$$AX * B + CX * D - / \quad (3)$$

which, without a need for parentheses or precedence relations, has only the interpretation of expression (2). With one more single scan across the string, it can be compiled into machine code.

The arithmetic scan described above is a special case of a general syntactic analyzer that uses precedence relationships.

REFERENCES

- 197 1. Ralston, A. *An Introduction to Programming and Computer Science*. New York: McGraw-Hill.

J. A. N. LEE AND A. RALSTON

ARPA NETWORK

ARPA NETWORK

For articles on related subjects see **COMPUTER NETWORKS**; **DATA COMMUNICATIONS**; **INTERFACE MESSAGE PROCESSOR**; and **TELEPROCESSING SYSTEMS**.

For article on related term see **HOST SYSTEM**.

In 1968 the Advanced Research Projects Agency (ARPA) of the US. Department of Defense embarked on a project to implement a nationwide computer network, called the ARPA network, or simply ARPANET. This network allows a large number of dissimilar computers, called "hosts," to communicate with each other. The major goals of the network are to

1. Permit computer resource sharing whereby programs, data, storage, special purpose hardware, etc., can be shared among many computers and users.

2. Develop highly reliable and economic digital communications.

3. Permit access to unique and powerful facilities that are economically feasible only when widely shared.

In late 1975 the network consisted of about 60 nodes and was expanding at the rate of about one node every three months.

Network Properties. Each host is connected to the network via a small local computer called the Interface Message Processor (IMP), shown in Fig. 1. Each IMP in turn is connected to several other IMPs via 50 kilobit/sec communication lines. Terminal IMPs (TIPs) with flexible terminal-handling capabilities are also available to provide a wide variety of terminals direct access to the network. Fig. 2 shows the geographic and logical maps of the network plus the computers at each node, and Table 1 shows the site abbreviations. Note that the network is multi-connected; i.e., more than one path exists between any pair of nodes. The topology was selected to provide good response times and high reliability, and have good growth potential while keeping costs to a minimum.

Message switching rather than circuit switching is used to establish communication between nodes. In a circuit-switched network, the source and destination are connected by a dedicated communication path established at the beginning of the

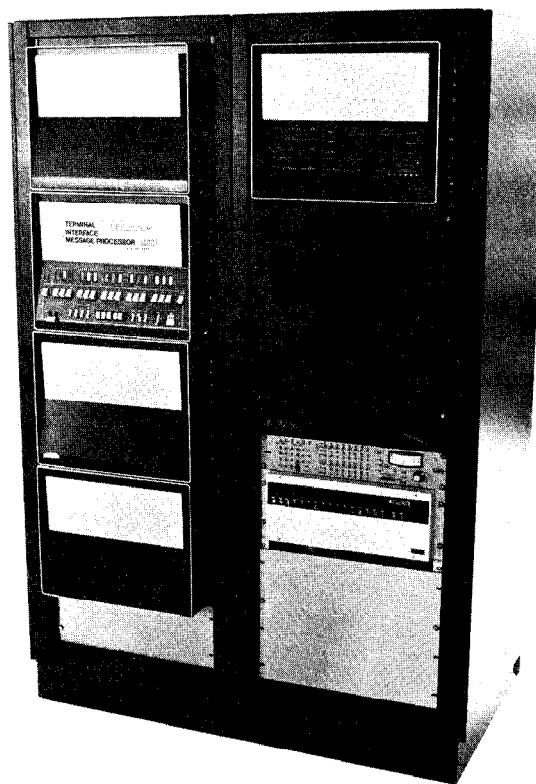


Fig. 1. A terminal interface message processor.

connection and broken only at the end. In a message-switched system such as ARPANET, no dedicated path exists. Instead, a source host or terminal passes its message, including a destination address, to its local IMP or TIP. The message is then passed from IMP to IMP until it finally arrives at its destination. The choice of the path is determined dynamically. Each IMP forwards the message on the path it determines best to assure prompt delivery, taking into account the network loading and failures. The current design allows variable length messages of up to 8095 bits. To improve transit times, the IMPs partition messages into 1024-bit packets and send them one after the other. Since more than one path exists between any two hosts, it is thus possible that the individual packets of a given message actually travel by different paths, depending upon the loading at that particular time.

Three steps have been taken to insure reliability. These include a multiconnected network, a 24-bit cyclic redundancy check, and an IMP that is

ruggedly constructed for protection against external environmental conditions. The cyclic check is used for error detection, with correction by retransmission. This reduces transmission errors to less than one bit in 10^{12} . Because of the multiconnections between IMPs, a single line failure will not isolate a node, nor will it prevent the flow of messages through the network. At least two adjacent line failures or an IMP failure are required to isolate a node. Once again, this will not prevent all other nodes from using the net.

The capacity of the network is the throughput rate at which saturation occurs, and is a function of the topology and capacity of the transmission lines, the distribution of traffic, and the average size of blocks being sent. The capacity can be easily improved by adding additional lines or upgrading some lines from 50 to 230.4 kilobits/sec (TELPAC C channel). To enable interaction, the target is that the transit time from any node to any other node for a 1000-bit packet should be less than 0.5 sec, with an average of 0.2 sec.

Special Network Centers. In any distributed system such as the ARPANET, it is difficult to detect failures quickly. For this reason, a special Network Control Center (NCC) has been established at Bolt, Beranek, and Newman to continuously monitor line or IMP failures and the volumes of host and line traffic. The IMPs are also equipped with automatic reloading facilities. This enables the NCC to reload any IMP with a copy of the operating program in case of revisions or an IMP memory crash.

There is also a Network Information Center (NIC) at Stanford Research Institute. This is a powerful on-line system, implemented on a PDP-10 computer, to provide ARPANET users with information on hardware and software resources available at each node of the network.

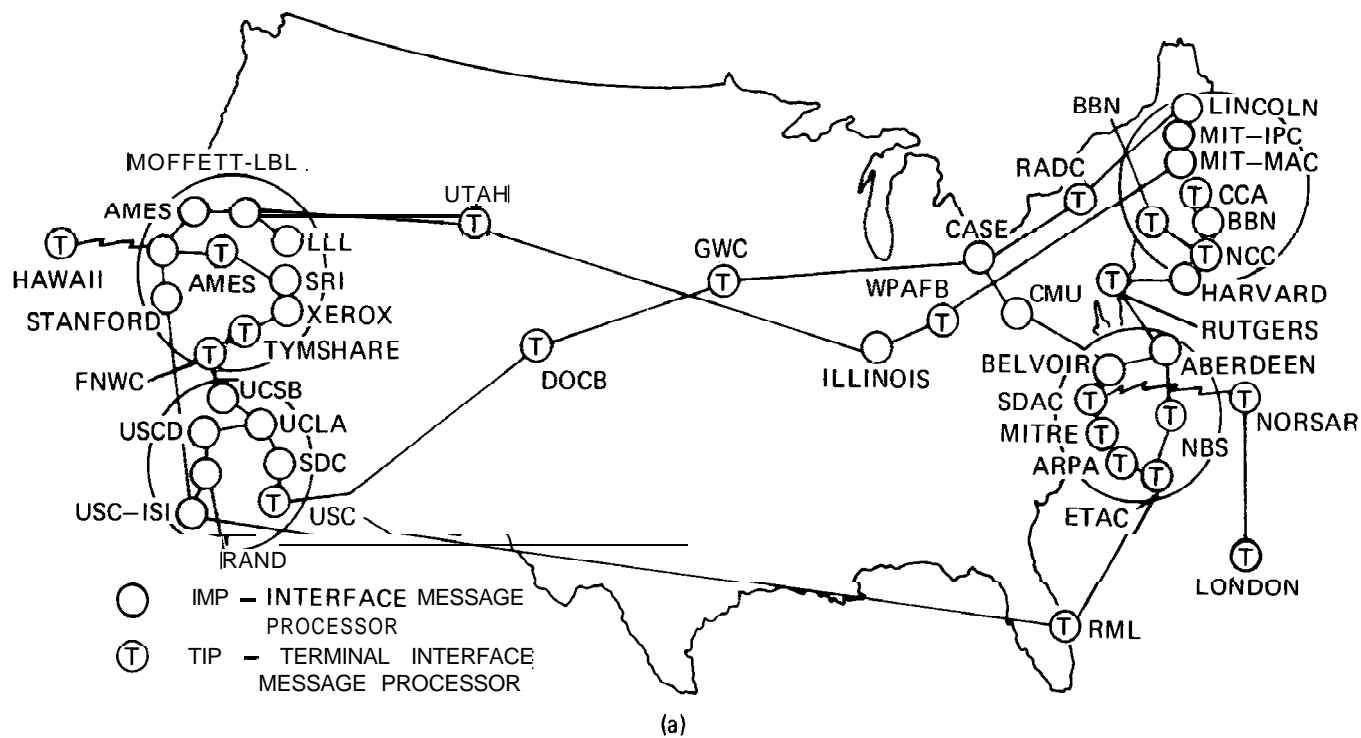
The Future of ARPANET. ARPANET has proved that message-switched networks are well adapted to interconnection of diverse computers and terminals. Since the network is operational and growing, it is conceivable that it will shift from a government-supported research and development activity to some national service organization. During its development, many vital questions concerning large computer resource-sharing networks have been answered. Nevertheless, technical improvements in the existing network will continue.

Projects currently under way include the development of IMPs capable of coping with data rates in

Table 1. Site Abbreviations

ABERDEEN	Aberdeen Research and Development Center
AMES	NASA Ames Research Center
ARPA	Advanced Research Projects Agency
BBN	Bolt, Beranek, and Newman
BELVOIR	Fort Belvoir, USAMERDC
CASE	Case Western Reserve University
CCA	Computer Corporation of America
CMU	Carnegie-Mellon University
DOCB	Department of Commerce, Boulder
ETAC	Environmental Technical Applications Center, USAF
FNWC	Fleet Numerical Weather Center
GWC	Global Weather Center
HARVARD	Harvard University
HAWAII	University of Hawaii
ILLINOIS	University of Illinois
LBL	Lawrence Berkeley Laboratory
LLL	Lawrence Livermore Laboratory
LINCOLN	MIT Lincoln Laboratory
LONDON	University College, London
MIT-IPC	Massachusetts Inst. of Technology, Inf. Proc. Services
MIT-MAC	Massachusetts Inst. of Technology, Project MAC
MITRE	MITRE Corporation
MOFFET	Moffet Field, USAF
NBS	National Bureau of Standards
NCC	Network Control Center at BBN
NORSAR	Norwegian Seismic Array
RADC	Rome Air Development Center
RAND	Rand Corporation
RML	Range Measurement Lab., Patrick AF Base
RUTGERS	Rutgers University
SCRL	Special Communication Research Laboratory
SDAC	Seismic Data Array Center
SDC	Systems Development Corporation
SRI	Stanford Research Institute
STANFORD	Stanford University
TYMSHARE	Tymshare Corporation
UCLA	University of California at Los Angeles
UCSB	University of California at Santa Barbara
UCSD	University of California at San Diego
UNIVAC	Univac Corporation
USC	University of Southern California
USC-ISI	University of Southern California, Infor. Sciences Inst.
WPAFB	Wright Patterson AF Base
XEROX	Xerox Corporation

excess of 1 million bits/sec and a study of the feasibility of using satellite communication channels. Problems include the type of operational procedures large networks should follow and the way in which programs and widely different operating systems should communicate with each other. Answers to



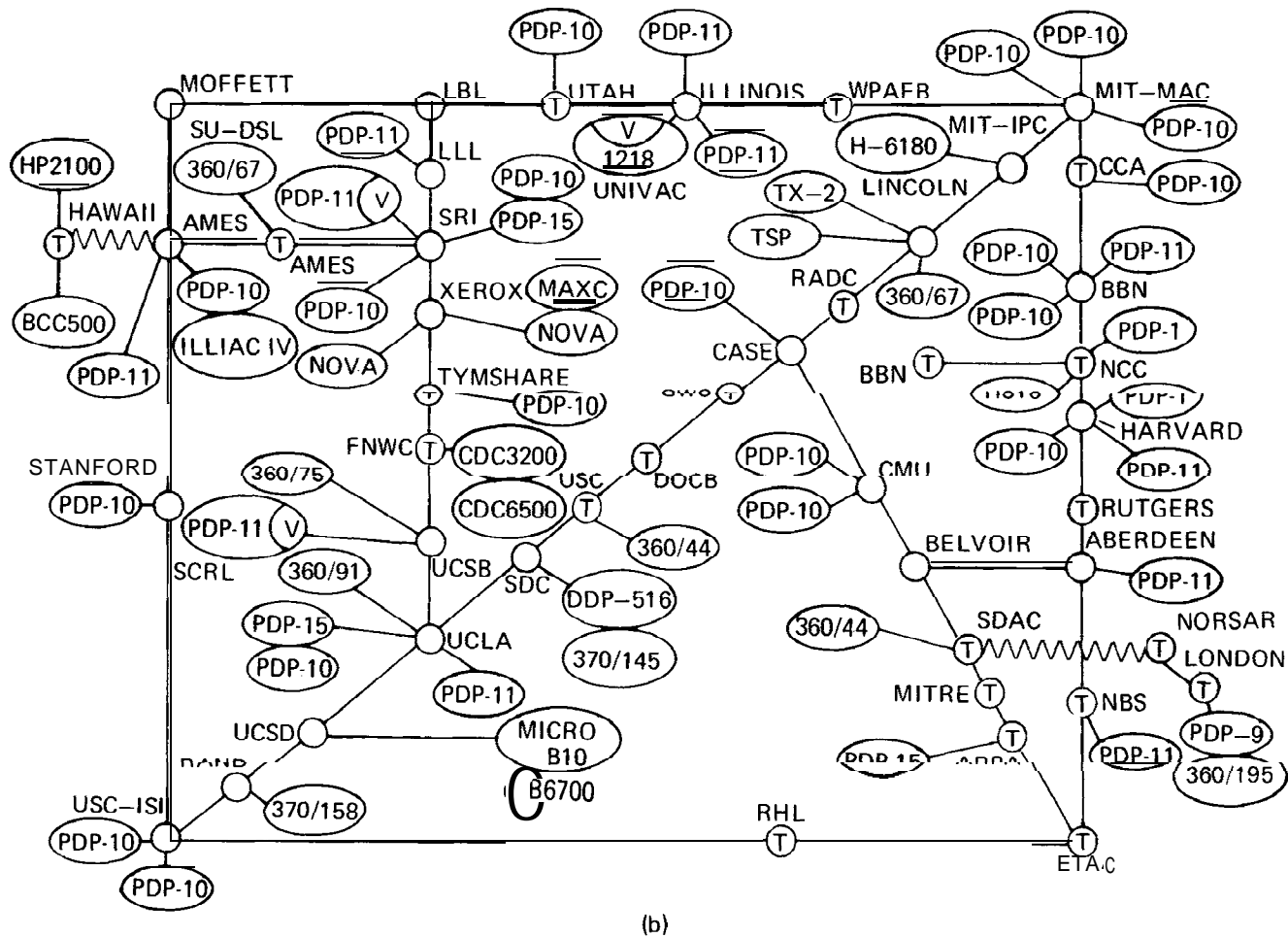


Fig. 2. Geographical (a) and logical (b) maps of the ARPA network issued by the ARPA Network Information Center (1974).

ARTIFICIAL INTELLIGENCE

these problems will lead to the improved use of computers.

REFERENCES

1970. "Resource Sharing Computer Networks," *Proceedings AFIPS Spring Joint Computer Conference*, pp. 543-597. (Five papers.)
1972. "The ARPA Network," *Proceedings AFIPS Spring Joint Computer Conference*, pp. 243-303. (Five papers.)

J. S. SOBOLEWSKI

ARTIFICIAL INTELLIGENCE

For articles on related subjects see **CYBERNETICS; GAMES ON COMPUTERS; HEURISTICS; PERCEPTRON; and THEOREM PROVING.**

For articles on related terms see **GRAMMAR, REDUCTIVE; INFORMATION RETRIEVAL; MODELS; SIMULATION; and SYNTAX, SEMANTICS AND PRAGMATICS.**

Probably the most controversial area of study in computer science is what we attempt to describe here under the generic term "artificial intelligence." This is understandable, since some of its assumptions, methods, techniques, and results are related to philosophical considerations and are associated with the nonexact social and life sciences.

One can often see references to machine intelligence, heuristic programming, simulation of cognitive, biophysical, evolutionary, etc., processes, self-organizing systems, or even to cybernetics in the same context. As we will see, some of this work could represent investigation as a subarea of artificial intelligence, or the terms might cover the same area as artificial intelligence itself.

We must have an idea of the substance of intelligence in general before we can talk about artificial intelligence. Without trying to give a precise and formal definition, a brief explanation would be that

... a system is judged to have the property of intelligence, based on observations of the system's behavior, if it can adapt itself to novel situations, has the capacity to reason, to understand the relationships between facts, to discover meanings, and to recognize truth. Also, one often expects an intelligent system to learn; i.e., to improve its level of performance on the basis of past experiences.

This loose but suggestive definition might be applied to nonliving systems or artifacts as well as to humans and animals. In fact, it must be stated that there is no scientific evidence that would limit, in principle, the inherent intellectual capabilities attainable by nonliving systems. Within a remarkably short period of time, interesting and significant results have been obtained in a number of different areas of artificial intelligence. A brief description of these is given in the following sections.

Approaches to and Objectives of Artificial Intelligence. It is useful to subdivide the whole area of artificial intelligence into two branches of study. One of these may be somewhat simplistically called the "engineering approach"; the other, the "modeling approach." In the first case, the researcher wants to create a system that is able to deal with interesting and difficult intellectual tasks, regardless of whether the methods and techniques used are similar or identical to those used by humans'. He has a job to accomplish inexpensively, efficiently, and reliably-that is all that matters. Examples of this approach are certain pattern recognition tasks (such as recognizing the characters printed in a special format on bank checks), translating text from one natural language into another, composing music by computer, locating warehouses across the country in an optimum manner, and so on.

The modeling approach has the basic research objective of trying to gain an understanding of the inside mechanisms of a real life system and to explain and predict its behavior. We can put in this category, for example, those projects that simulate human problem solving, decision making, or learning behavior by, for example, building models of neural networks.

There is, however, an area of transition between the two approaches, one that concerns certain intellectual tasks at which humans have been excellent over a long period of time. Chess, for example, has a written history of some 400 years and its rules have changed very little in many centuries. It has presented a challenge to researchers in artificial intelligence par excellence. About a dozen chess-playing programs have been reported in the literature and the more recent ones can play reasonably sophisticated games, not too much below the master level. Because of the impossibility of reducing this game to a mathematical formalism (i.e., it is not amenable to an algorithmic solution) and because an exhaustive method of finding the optimum move is in general out of the question, a good chess-playing program

has to incorporate so-called heuristics. These are based on loosely formulated rules of thumb that are occasionally referred to as "insight," "intuition," or simply "experience."

In other words, the knowledge of how humans perform certain tasks is indispensable to and must be incorporated into those programs whose main objective is to accomplish these intellectual tasks as well as possible.

We can now briefly enumerate the three basic motivations for research in artificial intelligence:

1. To replace human intelligence because the latter is expensive, scarce, and often less than reliable.
2. To establish theories of human intelligence in the form of simulation models.
3. To assess the capabilities of presently available software and hardware, and to point to lines of development for future programming languages and computer systems.

Three Fundamental Problems. In the following discussion a large number of study areas will be outlined. Three problems are common among all these areas and it is therefore advisable to examine them briefly at this stage.

PROBLEM OF REPRESENTATION. The selection and design of representation is a central issue in programming in general. How a particular task is translated into information structures and information processes inside the computer may render the solution of the task efficient and effective, or so cumbersome that it is prohibitively unwieldy.

One of the long-term objectives of artificial intelligence research is to automate the processes that make the decision on a particular representation after the specification of the task has been given, possibly in natural language. At present, however, the information structures and processes are chosen ad hoc and are assumed to be quasi-optimum for the task at hand. This task dependence in representation has obvious shortcomings.

GENERALITY VERSUS EFFICIENCY. Human intelligence is multipurpose by nature. The majority of projects in the study of artificial intelligence have, however, resulted in somewhat narrow and single-minded programs whose efficiency is reasonably high. As soon as the range of applicability of a program increases, its level of complexity rises, but its efficiency in solving individual problems drops.

Again, a long-term objective is to write highly efficient programs of "universal" applicability. A plausible avenue to this goal is via learning programs

that can initially tackle simple problems and gradually acquire more and more power for a larger number of, and more difficult, problems. Also, a reasonable level of success has already been achieved with programs that subdivide a large and difficult task into smaller, possibly known ones. This technique has to be developed further.

PROBLEM OF SEARCH. It is often the case that one can write a program that generates potential solutions to a problem, tests them, and, hopefully, sometimes discovers the right ones. With nontrivial problems, however, the "solution space" is very large, for practical purposes infinite, and there can be no exhaustive search performed. It has been, for example, estimated that in order to find the best possible starting move in chess, the machine would have to evaluate 10^{120} game positions. It has also been pointed out that if a computer consisting of all the elementary particles in the universe could be constructed and run with the speed of light, the current estimate of the age of the universe would not have been long enough to find the best first move according to the exhaustive search strategy.

It would therefore be a good idea to direct the search process by detecting relative improvements as the program moves around in the domain of all potential solutions. Heuristic rules should be built in or, in a more sophisticated manner, automatically generated by the program to recognize the structure of the search space and thereby cut down the computing time and memory requirements. Although many efficient search techniques have been developed in various studies in artificial intelligence, there is a great need for further work in this area.

Finally, it should be noted in a nonapologetic manner that the area discussed, like the majority of the branches of computer science, has a predominantly experimental flavor. Although more and more theoretical foundation is being developed in several directions, the basic motive of experimentation is likely to prevail in the foreseeable future.

Research Topics in Artificial Intelligence.

Mechanical and electromechanical toys have been built for a long time to demonstrate certain goal-oriented behavior. More recent ones (by Shannon, MacKay, Ashby, etc.) have shown certain *cybernetic* principles involving negative feedback concerning, for example, the recharge state of their batteries. The robot projects currently pursued at Stanford University, Stanford Research Institute (see Fig. 1), M.I.T., and various Japanese and British laboratories, go well beyond them in sophistication. The robots not only can perceive visual stimuli via

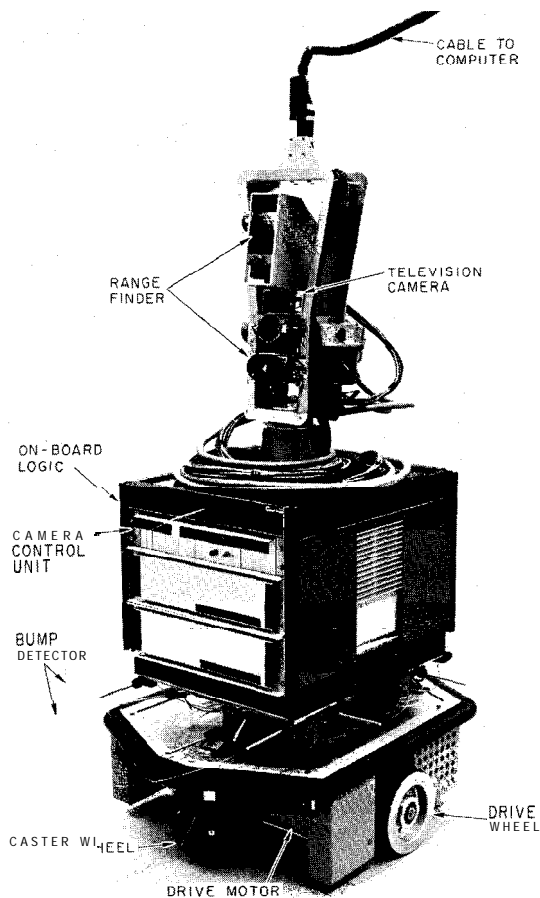


Fig. 1. "Shakey," a robot project of the Artificial Intelligence Center of Stanford University.

television cameras but are also equipped with tactile receptors and some effectors to move, for example, wooden blocks around. The controlling computer may communicate with the robot via cables or radio signals, or the robot may carry the computer along its adventurous path.

These projects aim at integrating the many piecemeal results of artificial intelligence research, from theorem proving to pattern recognition and picture processing, from learning and problem-solving techniques to natural language processing and to question-answering systems. Beyond the interest in basic research, workers in this field hope to make use of their results in planetary or underwater explo-

ration and in areas where human access is difficult or impossible.

Simulation of neurological and physiological phenomena often consists of building a computer model of a network of (idealized) nerve cells. These networks are self-adaptive; i.e., they can be trained to recognize audio and visual patterns by changing certain parametric values or the interconnections between them. One of the most remarkable of these experiments was made by Rochester, Holland, Waibt, and Duda to test Hebb's verbally described theory of neural cell assembly. They discovered some flaws in the assumptions, and after making some extensions of the model, they were able to show the learning capability of the network.

Rosenblatt and his coworkers investigated a class of random nets of neuron-like elements, called "perceptrons." Their mathematical analysis was recently extended by Minsky and Papert. The EEG-like output of permanently connected nonlinear elements was demonstrated by Farley. Wooldrige and Broadbent tried in various studies to describe the activity of the brain in terms of electric circuitry.

Another idea of somewhat limited success by Fogel and others was to simulate the evolution of intelligence from its most primitive beginnings via mutations and natural selection.

Several projects have simulated robots in environments of varying complexity (Doran, Toda, Findler, and Allan). Planning and search behavior have been interesting aspects of these works. The problems of heuristic search have often formed the basis of some abstract investigations (Sandewall, Pohl, Nielsson), sometimes in connection with the special task of the Graph Traverser (Michie, Doran, Marsh, Ross). Techniques of tree search are of primary interest in game playing and theorem-proving programs.

The simulation of cognitive processes will now be discussed in some detail. (Affective processes involving motivation, ambition, etc., have been considered by Simon, Findler, and others.) The basic tenet here is that man's complex behavior can be analyzed and broken down into elementary symbol manipulation processes. The computer can be programmed in terms of these elementary building blocks to perform the same information-processing tasks that humans do. The computer program, representing a psychological theory of various behavioral phenomena, becomes a convenient vehicle to determine objectively the implications of the model. The need for precise formulation in the program leaves no room for vagueness, ambiguity, and lack of rigor. A comparison with experimental

data may lead to modifications in the model until the researcher is satisfied that he has constructed a sufficient theory. According to Newell and Simon, human thinking can be explained in information processing terms without waiting for a theory of the underlying neurological mechanisms.

An often used technique for finding out about the details of human thought processes is to make the experimental subject utter the reasons behind every step of his activity verbally into a tape recorder ("thinking aloud"). A detailed analysis of these "protocols" would lead to a flowchart of the subject's behavior. (Newell and his students have obtained promising results in trying to automate the process of analysis.) The flowchart is then translated into a program, several variants of which can be run to simulate different individuals.

Three classical pieces of work along these lines were produced by Newell, Shaw, and Simon. The first one, the *Logic Theorist*, is a heuristic theorem-proving program for the first-order predicate calculus. Its original version could prove 38 of the 52 theorems in Chapter 2 of Whitehead and Russell's *Principia Mathematica*.

The second work, the *General Problem Solver* (GPS), has been able to separate to a significant extent problem-solving techniques and task environments. The so-called means-ends analysis lies at the center of the program. It goes as follows: The present and the desired objects are given ("object" in the most general sense of the word). GPS discovers the difference between them and tries to find an operator (i.e., a transformation) relevant to this difference. If successful, the task is accomplished. Otherwise, it attempts to bridge over the much too large difference by creating objects in between the present and the desired objects. In this recursive manner, either it produces a sequence of operators that is the solution of the problem or it reports failure. The latter may also be due to the fact that the program has exceeded the prespecified time limit or has run out of memory. This basic paradigm of problem solving has penetrated many projects, including the Newell, Shaw, Simon chess program, which is the third piece of work referred to above.

Concept learning has been the target of several simulation models (Hunt, Holland, Johnson, Baker, and others). The behavior of these programs reflects the attribute and rule-learning characteristics of humans. The detection of temporal sequences implies an understanding of event generation in order to predict future members of the sequence. Simon and Kotovsky, Abrahams et al., D. S. Williams, and others have written programs to simulate humans in

these tasks. The first of this kind of program was in fact Feldman's *Binary Choice* Model, which reproduced even the idiosyncratic behavior of several individuals.

The elementary perceiver and memorizer (EPAM) by Feigenbaum and Simon (1963) represents a theory of human memory and models the learning of associated nonsense syllables, a standard psychological test. It reproduces the increasing discriminative ability, retroactive inhibition, stimulus generalization, response oscillation, and other phenomena usually pinpointed and analyzed by psychologists conducting tests on experimental subjects.

Two other complex information-processing models also have an aspect of practical applicability. Tonge has written a heuristic program that balances factory assembly lines (to keep idle time of machines minimum or throughput maximum), and Clarkson's simulated financial advisor selects a quasi-optimum portfolio for investors.

In his study of human decision making under uncertainty and risk, Findler experimented with and simulated subjects that optimized certain state variables by selected control-variable values in a dynamic task environment. In a later work, Findler, Klein, and their students studied the generation and utilization of heuristic rules within the framework of the card game poker.

Evans' program is able to discover geometrical analogy between line drawings. Although formula-manipulating languages are outside the domain of this article, the heuristic formal integration programs of Slagle and Moses must be mentioned here.

Another program of great potential applicability is by Feigenbaum, Lederberg, and others, called the "heuristic dendral," which discovers the most likely molecular configuration of certain chemical compounds on the basis of spectroscopic data and built-in chemical knowledge. F. A. Miller constructed a practical heuristic regression analysis program that improves with experience in a particular problem.

Individual belief systems have been simulated by Abelson and Carroll. Colby has worked on computer models of neurotic patients. Weizenbaum's ELIZA program can imitate a certain type of psychiatric interviewer.

McCarthy and his students proposed a system called "Advice Taker," which is more formal than GPS but has similar objectives in making "common sense" deductions from facts. Interpersonal interactions were modeled by Gullahorn and Gullahorn, following Homan's sociological theory. McWhinney

ARTIFICIAL INTELLIGENCE

studied and simulated human communication network experiments.

Loehlin worked on a program that reproduced individuals' likes and dislikes, depending on a complex, interacting set of attributes. Findler and **McKinzie** wrote programs to simulate demographical processes, and to build and query complex kinship structures from primitive input information.

Research on intelligent question-answering programs has become very popular in the past few years. These programs not only retrieve information stored in the computer memory but also make logical inferences and (may) ask the user for more information in case of ambiguity. **Bobrow's** **STUDENT**, for example, solves high school algebra problems stated in much restricted English. **Simmons** and his coworkers have produced several question-answering systems of varying sophistication, from which the **Protosynthes** projects ought to be singled out for their very large data base.

The basis of most of these projects is a *semantic network*, which consists of concept nodes connected by association links. **Raphael's** Semantic Information Retrieval program, **Coles' Picture Language Machine**, **Black's** Deductive Question-Answering System, and **Quillian's** Semantic Memory and Teachable Language Comprehender must be mentioned in this context. The problems of grammar of natural languages have been tackled by modern theories of syntax (by **Chomsky** and others) reasonably successfully, but *meaning* has eluded satisfactory treatment. Semantics has been shown to be the central problem in the study of language. One needs to think of the resolution of ambiguities only to appreciate the importance of it. Machine translation is unlikely to reach a significantly higher level of success until computer programs can exhibit understanding in the full sense of the word. **Winograd's** work in this area shows great promise. His system, which does not separate syntax and semantics, simulates a robot that follows instructions given in an interestingly rich subset of English that reasons, that asks questions of and provides answers to an interactive user about the small universe of wooden blocks of different colors and sizes. Information retrieval has become an endeavor in its own right. Many of its techniques, however, are closely related to those used in the area under discussion.

Interesting work is going on in automatic speech recognition and synthesis (**Reddy**, **Hill**, **Vincens**, and others). A task force has been set up to specify the work to be accomplished in the next five years. This refers to the vocabulary size of recognizable words,

the quality and number of different speakers, the tolerable error rate, etc.

Final Comments. We have given a necessarily cursory overview of the major research areas in artificial intelligence. Some topics were deliberately left out, such as game playing, pattern recognition, image and picture processing, computer assisted instructions, augmentation of human intellect, computer-aided design, management information systems, social science and humanities applications, language translation, arts and medical applications, each of which is closely related to and overlapping our interest and is discussed in other articles in this volume.

Two more issues should be discussed here briefly. First, one often hears the statement of the skeptic: "A computer can only do what it is told to do." If this is so—and no computer professional has ever doubted the plain truth of this saying—how can we expect intelligence to emerge from our *machines* if they simply follow our instructions? The answer lies in the interpretation of the statement. Admittedly, computers execute the orders in a program step by step. However, with every nontrivial program (and we are talking about extremely complex hierarchies of programs), its creator does not know what the intermediate or final results will turn out to be. There is no way of telling how a program will behave under any of a large, possibly infinite number of conditions it will be exposed to. The situation is not dissimilar to the case in which the parents and teachers of a child cannot, with any certitude, foretell how he will act later in his life, given the hereditary, educational, and environmental information up to a certain point in time. Learning programs, although still somewhat in their infancy, often outperform the persons who wrote them.

The other side of the coin can be seen when a particular program that exhibits some high-level intellectual capability is described in terms of the elementary processes on which it has been built. Then our respect for the program suddenly drops because, as **Minsky** puts it, the problem is "explained away." Thus, many people will no longer consider that particular task to require intelligence. Human thinking may meet with this fate sometime, too.

The second issue is called the "superhuman fallacy." Many projects in artificial intelligence receive a paternalistic, deprecatory criticism: "Well, that computer-composed music is pretty awful"; or, "It is all right to prove those theorems in Euclidean geometry but the machine surely could not have *invented* Euclidean geometry"; and so on. Let us ask,

however, how many Mozarts, Euclids, Einsteins, or Shakespeares have there been? We find only one of each, and each has been the descendant of the generation after generation development of human intelligence. We must compare this with only 15 years' progress in artificial intelligence research and development to obtain an approximate perspective.

SUGGESTED READINGS

The literature in artificial intelligence is naturally scattered over many books, conference proceedings, and journals. The following provides a first approximation to the most important subset of relevant literature and contains discussions of or references to almost all the projects discussed in this article.

Books

- 1967-1972. Meltzer, B., and D. Michie (Eds.). Articles by N. L. Collins and D. Michie; E. Dale and D. Michie; D. Michie; B. Meltzer and D. Michie; B. Meltzer and D. Michie in *Machine Intelligence*. Edinburgh, Great Britain: Edinburgh University Press. (These are collections of most relevant papers presented at one of a series of annual workshops. So far, seven volumes have been published.)
1968. Minsky, M. (Ed.). *Semantic Information Processing*. Cambridge, Mass.: M.I.T. Press. (This is a collection of Ph.D. theses plus contributions by Minsky and McCarthy.)
1971. Findler, N. V., and B. Meltzer (Eds.). "Artificial Intelligence and Heuristic Programming." Edinburgh, Great Britain: Edinburgh University Press. (This is the *Proceedings of the First Advanced Study Institute on Artificial Intelligence and Heuristic Programming*, with 14 contributions.)
1971. Feigenbaum, E. A., and J. Feldman. *Computers and Thought*. New York: McGraw-Hill. (Contains some classical contributions to the area up to 1962.)
1971. Slagle, J. R. *Artificial Intelligence-The Heuristic Programming Approach*. New York: McGraw-Hill. (This is a more up-to-date introductory survey of the main research projects.)
1974. Jackson, P. C., Jr. *Introduction to Artificial Intelligence*. New York: Petrocelli/Charter. (This is a rather comprehensive description of research projects in artificial intelligence and in areas closely related to it.)

Conferences

- There were two *International Joint Conferences on Artificial Intelligence*, the first in Washington, D.C., 1969 and the second in London, England, 1971. The proceedings are full of excellent modern papers.
- The *Proceedings of the IFIP Congresses* (Paris, 1959; Munich, 1962; New York, 1965; Edinburgh, 1968; Ljubljana, 1971) contain several interesting contributions to the field. Similarly, the yearly *Spring* and *Fall Joint Computer Conferences*, and the annual *ACM National Conferences* always publish papers in artificial intelligence in their respective proceedings. There are many other meetings in related areas every year-it is impossible to list them here.

Journals

- International Journal of Man-Machine Studies*, *Information Sciences*, *International Journal of Computer and Information Sciences*, *Artificial Intelligence*, *Behavioral Science*, *ACM Communications and Journal*, *Computer Journal*, *Kybernetik*, *Cybernetica*, *IEEE Transactions on Computers*, *Systems Science and Cybernetics*, *Information and Control*, etc.

surveys

- Of the many survey papers of varying age, three are singled out.

1961. Minsky, M. "Steps toward Artificial Intelligence," *Proc. IRE*, Vol. 49, pp. 8-30. (Reprinted in the book edited by Feigenbaum and Feldman, 1971.)
1963. Newell, A., and H. A. Simon. "Computers in Psychology," in Luce, Bush, and Galanter (Eds.), *Handbook of Mathematical Psychology*, Vol. I. New York: Wiley.
1968. Hunt, E. B. "Computer Simulation: Artificial Intelligence Studies and Their Relevance to Psychology," *Annual Rev. of Psych.*, Vol. 19, pp. 135-168.

N. V. FINDLER

ARTS APPLICATIONS

For articles on related subjects see **COMPUTER GRAPHICS**; **HUMANITIES APPLICATIONS**; **IMAGE AND PICTURE PROCESSING**; **INFORMATION RETRIEVAL**; and **PATTERN RECOGNITION**.

For article on related term see **DATA BANK**.

As is the case with computer applications in the humanities, computer applications in the arts (painting, sculpture, film making, music, and dance) include the use of data banks in pattern recognition (analysis) and pattern generation (synthesis).

Data Banks in the Arts. Just as libraries have traditionally served as data banks for the verbally oriented humanities, museums have traditionally served as data banks for the nonverbally oriented humanities. Efforts to use the computer to make museum collections more accessible to users have resulted in the development of a number of data bank projects for museums. Examples of such projects are GIPSY (General Information Processing System), used to record ethnographic museum specimens in Oklahoma, Missouri, and Arizona; GIS (General Information System), used in the FLORA N. A. program, a large-scale, centralized data bank designed to collect, analyze, maintain, and disseminate diverse kinds of information about the plants of North America; GRIPHOS (General Retrieval and Information Processing for Humanities Oriented Studies), used by the Museum Computer Network, an organization of several dozen museums and related organizations, primarily to record art objects and archeological data; SELGEM (Self-Generating Master), used to record specimen inventories of mammals, conodont types, foraminifera, nematodes, crustacea, and other collections; and TAXIR (*Taxonomic* Information Retrieval), used primarily for the storage and retrieval of biological specimen data (see Chenhall et al., 1972).

In the United States, the development of these systems has promoted the formation of a museum data bank coordinating committee (Chenhall et al., 1972), among whose functions are the following:

1. To develop comparative descriptions of the general information systems that are presently available so that a potential new user would have an objective basis for deciding which system was most appropriate to his needs.

2. To serve as a clearing house of data categories and minimal standard recording conventions for all museum data banks, so that data recorded in one of the systems will be compatible with that recorded in other systems. Standard recording conventions would be recommended only for such categories as dates, proper names, etc., or when requested by a representative body of scholars for a particular discipline.

3. To serve as a central point for the communication of information to and from other data bank organizations around the world.

4. To coordinate the development of programs for the conversion of data from one system to another.

5. At a later date, when sufficient information has been gathered in data banks across the country (U.S.), to coordinate or contract for the synthesis of actual data for specific disciplines on a regional or national basis.

These goals of the committee indicate the problems attendant in the use of massive data banks.

Other types of information banks necessary for pattern recognition and pattern generation in the arts include bibliographies, special-purpose catalogs, musical scores in computer-accessible form, and rules for combining elements and artifacts.

An example of a bibliographical project that includes abstracts as well as the standard author, title, etc., reference information is the *Répertoire International de la Littérature Musicale* (RI LM). This computer-based information bank currently references scholarly publications on music from the year 1967 onward. After production problems are solved, the plan is to move backward from the year 1967 (while, of course, continuing to keep up with current publications) so as to provide in time a comprehensive computer data bank on publications in music (see Brook, 1967). An example of a special purpose catalog (in music) is that proposed for French chansons, which would include a listing of all manuscript and printed sources for chansons, musical incipits (notes at the beginning of the chansons), a melodic index, and other relevant information (see Hudson, 1970).

Information banks consisting of musical scores are requisite for pattern recognition studies focused on either a particular composer or a musical genre. No extensive data bank of musical scores has yet been provided, primarily because of the input problem. To date, almost all computer scores that have been put into computer-accessible form have been translated into Teletype keyboard characters using

either the Ford-Columbia system of transcription (DARMS) developed by Stefan Bauer-Mengelberg of the State University of New York at Binghamton or the Plaine and Easie Code (see Brook and Gould, 1964), or systems developed by various individual scholars for their own specific research needs.

An alternative to transcribing musical scores into keyboard characters is the use of graphic input devices. One such experiment displays (on a cathode-ray tube) a musical staff as well as conventional notation for musical notes, pauses, and other standard music symbols. The scholar may then select the appropriate symbols and place them as desired on the music staff. Although this system has been proposed for music editing, it is clear that such an approach could be used to input scores, once some of the limitations (e.g., the availability of upward stems only) of the experimental system were eliminated (Cantor, 1971). The use of graphic input devices has also been advocated for data bank information necessary to computer generation of music.

An approach devised somewhat earlier than the one mentioned above considered the cathode-ray tube display as a grid across which the composer could draw graphlike lines to indicate amplitude, frequency, note duration, and so on (Mathews and Rosler, 1969). Because this particular graphic input system did not use common music notation, the user had to learn a notational system in order to input his data. Thus, the only presumed advantage of this system over transcription into keypunch characters would be the greater speed with which data could be input.

Another important type of data bank for both pattern recognition and generation in music is digitized sound. Breaking down musical acoustical signals into digitized form as well as using the digitized form to generate acoustical signals requires an analog-to-digital and digital-to-analog converter system with a speed sufficient for 20,000 to 50,000 samples per second and a quantization accuracy of from 10 to 13 bits. As is the case for speech, storage of digitized music poses space requirements that are sometimes a seriously limiting factor on the scope of the research being undertaken (Beauchamp, 1967).

Pattern Recognition. Pattern recognition projects focused upon painting and sculpture are almost nonexistent. Thus far, fine arts museum information banks consist of listings of artists, works of art, dates when created, perhaps the date of acquisition by a given museum, and other comparable detail. Although there is a strong interest in

providing content information (e.g., physical objects represented in a painting; the colors used in a painting), there is at present little effort to make such information available as part of computer-accessible information banks. Conceivably there will be some effort to hand-code such information, but any such encoding, albeit useful, will be prone to the subjective responses and lapses of any given encoder. It seems likely that in time (whether measured in years or decades is not yet clear), pattern recognition efforts currently being pursued in other areas (e.g., aerial photography, artificial intelligence research on robots, character recognition) will be used to facilitate input of information concerning museum holdings.

Current work on identification of two- and three-dimensional objects, on methods of locating and following edges so as to identify objects, and on the use of regional analysis rather than edge delineation to describe objects is all obviously relevant to pattern recognition of paintings, sculpture, and other objects in museums (Guzman, 1968; Rosenfeld, 1969a,b). A recent dissertation by Lawrence Krakauer, M.I.T., on computer analysis of visual properties of curved objects uses digitized photographs of apples, oranges, peaches, Bartlett pears, and sweet pears as data banks. Mathematical and statistical techniques are then used to identify and describe parameters related to shape (e.g., stems, protrusions, stem hollows) and texture (visual texture and tactile texture) so as to describe and discriminate among the types of fruit being analyzed.

The determination of characteristics sufficient to identify a few known artifacts is hardly comparable to digitizing and then describing the artifacts in even the smallest of museums. It also should be noted that the work on the fruit entailed only black and white images, although the addition of color apparently poses no serious additional problems. It may be that it will be many years, if ever, before it is feasible and practical to try to map categories such as boat or vase (let alone Etruscan vase as opposed to Grecian vase) onto parameters indicating shape (such as eccentricity), or texture, or alignment of edges, or whatever indicators emerge from digitized representations of artifacts. Nonetheless, we may expect the parameters that do emerge easily from such digitized representations will in themselves provide useful categories for the art historian or student of style and content in painting and sculpture.

In contrast to painting and sculpture, pattern recognition in music is a very active research area. Pattern recognition involving digitized acoustical

ARTS APPLICATIONS

signals faces the storage problem mentioned earlier. For example, an effort to analyze tones produced by a range of musical instruments used a hardware configuration that did not permit adequate continuous transfer of digitized analog samples to magnetic tape. As a result, only core storage was available in this particular configuration and (using 29,952 samples per second at a sample rate of 30 kc) the maximum length of sound that could be digitized was 0.997 sec. This block of information could, of course, then be transferred to tape and a new 0.997 sec of acoustic signal could be stored and transferred. These blocks of data were characterized and graphed to show time-variant spectral information (Beauchamp, 1969).

In general, work on digitized music can deal with such parameters as frequency or pitch, amplitude, timbre, tone, and phase. When storage and processing speed problems are solved, there will probably be music information banks consisting of digitized music as performed, rather than, or in addition to, its appearance in a printed score. If and when such data banks are available, it may well be analogously to possibilities for museum artifacts—that the parameters used for characterizing music will not all be mapped onto the traditional symbols of a printed score. In the case of music, as well as in spoken and written language, what could eventuate is a parallel development involving, on the one hand graphic pattern recognition devices and, on the other, acoustic pattern recognition techniques.

Much pattern recognition in music has depended upon information banks consisting of musical scores. Representation systems, such as the Plaine and Easie Code or DARMS, permit the scholar to encode in keypunch characters all the symbols in a written score. Written scores permit studies of styles of individual composers as well as groups of composers. Written scores cannot be used for studies of interpretations of a given work by conductors or performing artists; digitized acoustical recordings might have to be used for pattern recognition of this type.

Computer-accessible music scores have been used for work on questions of harmony, of composer identification, of particular themes or motives within the works of a particular composer, and for ethnomusicology. Many of these studies concentrate upon pitch and combinations of pitches, although other information such as duration of notes, amplitude of notes, and various frequency distributions of notes is obviously available from music scores. For example, studies concerned with harmony depend upon

chords and patterns of chords, which are described in terms of the pitches of the individual tones combined into chords (Fuller, 1970; Jackson, 1970). A study of motif, or musical theme, again looks at the pitch of notes and combination of pitches into chords, but it is also concerned with changes of dynamics and tempo (Fiore, 1970). Studies of dynamics and tempo imply analyzing information concerning the amplitude (loudness-softness) with which the individual notes or chords are played and analyzing information about the duration of individual notes or chords. One project explicitly directed toward composer identification was concerned with root progression, which is a study of the way chords are connected-again, a study emphasizing pitch (Youngblood, 1970).

Work in ethnomusicology is directed toward describing the music of a particular culture or, sometimes, comparing the musical habits of a number of cultures (Lieberman, 1970; Suchoff, 1970). A project directed toward analyzing Japanese music concentrated upon fixed melodies (F.M.) and variations on the F.M. achieved through its rendition by different instruments and through elaborations played on still other instruments. Once this data was encoded, the computer was used to sort out and derive statistical summaries of the data (Lieberman, 1970). It should be noted that implicit in the pitch of any given tone is not only its location relative to any given octave but also the octave in which it actually occurs (e.g., an A in the octave above middle C).

Pattern recognition in dance is of considerable interest because the dance notation systems that do exist are relatively little used (Noll, 1967a). Hence, dances exist principally in the heads of choreographers and the dancers, and often a particular dance vanishes when its originating dance company breaks up or a choreographer passes from the scene. In the context of a discussion of generating dance movements by computer (Fig. 1), it has been pointed out by Noll (1967a) that specifying human movement in any detail is exceedingly complicated:

... obtaining the equations for as simple a motion as walking would be formidable. A better attack on this problem might be for the computer itself to analyze human motion using devices which have just become available for converting pictorial data into machine digestible data. A library of basic movements could be built within the computer, and particular movements could then be put together at will.

As should be apparent from the earlier discussion of painting and sculpture, the pattern recog-

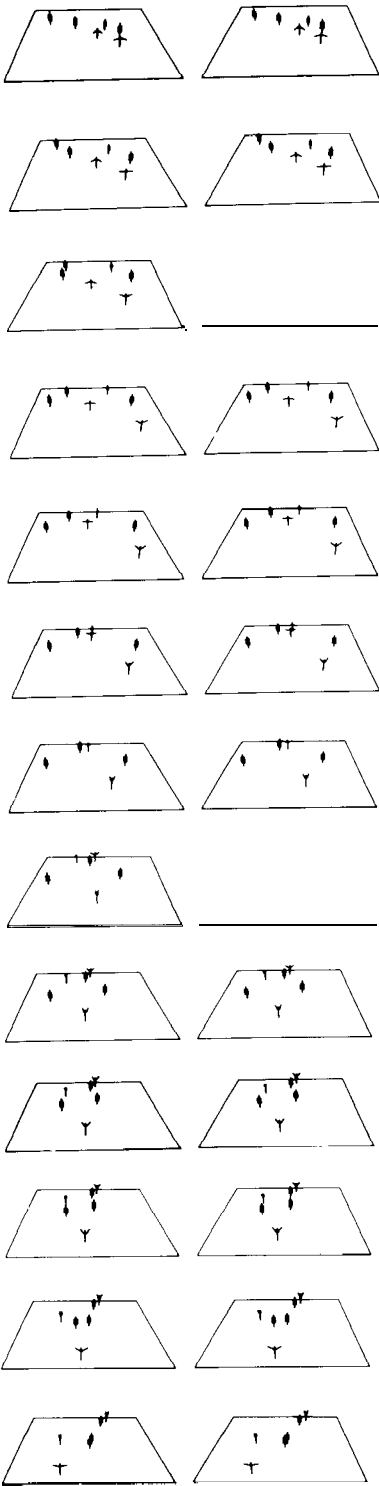


Fig. 1. Computer-generated ballet. Courtesy of A. Michael Noll.



Fig. 2. Computer composition with lines. Courtesy of A. Michael Noll. (© AMN 1965.)

nition problems implied by this approach are complex. Walking entails the angles of legs, feet, and other parts of the body, which are changing through time (see Fig. 2). As it is possible to imagine in the cases of painting, sculpture, and music that other than traditional parameters may be useful for pattern recognition, so, too, may be the parameters other than those to which the dancer is accustomed. Thus, the preparation of information banks to be used for pattern recognition that could take the place of or provide the stimulus for dance notation would seem to imply a mapping of whatever parameters are best for the digitized information onto those that are best for the dancer.

Pattern Generation. In contrast to computer-based pattern analysis, pattern generation in painting and sculpture and its allied field, film making, is firmly established. Computer-generated paintings and films in two, three, and even four dimensions have been produced. Output media have ranged from computer printers to cathode-ray tubes, and input media from punched cards to lightpens. One early approach (Csuri and Shaffer, 1968) which resulted in a so-called sine curve man and a prize winning film, *Hummingbird*, used the following technique:

- (1) an artistic drawing was made with line segments of points of the subject matter to be used;
- (2) a drawing was digitized line by line with the resulting coordinates punched into

ARTS APPLICATIONS

cards; (3) decisions were made about the type of form modification and the mathematical steps required to accomplish it; (4) the mathematical algorithm was programmed for a computer which then generated the plotter commands; (5) at this point, another decision was made about the color and line width for the transformation; (6) the transformed image was plotted on a Calcomp 563 plotter.

To produce a film from individual drawings, the figure of a hummingbird was transformed slightly in a succession of drawings, which were then filmed and projected in the conventional way to produce the illusion of movement.

Currently, graphics packages are available which permit an artist or film maker to sit down at a display console, specify the number of x - y coordi-

nates sufficient to define whatever figure he wishes to create, and then transform it according to the specifications made available in the graphics program. Three of the basic transformations are scaling, translating, and rotating. Scaling implies modifying the scale of all or some part of the figure that has been created; translating implies moving all or some part of the figure to another location on the output medium; and rotating implies altering the position of the object so as to make it seem to move around an axis as viewed from the perspective of the artist.

The art generated has ranged from very geometrical designs to highly random, apparently chaotic patterns (see Fig. 3). Some efforts have been made to approximate the painting styles of human artists (Canaday, 1970; Noll, 1967c).

Two-dimensionality is achieved simply through specifying x - y coordinates for a plane surface. Ac-



Fig. 3. Contained contour IX. Courtesy of Colette and Charles Bangert.

cording to Noll (1967b), a three-dimensional

... perspective drawing is produced by choosing a point (representing the eye and formally called the station point) from which the object is viewed. The picture plane is then inserted between the object and station point, and projection lines are drawn from the object to the station point. The points of intersection of these projection lines with the picture plane are joined together to produce the perspective drawing.

Four-dimensional objects can be specified mathematically and produced by mathematical formulas analogous to those used for two-dimensional perspectives of three-dimensional objects. Although it is impossible to see a four-dimensional object, the three-dimensional (and, in turn, two-dimensional) projection of the four-dimensional object produces effects that differ from conventional two- and three-dimensional artifacts. It has been noted that "the programs and mathematical techniques for four-dimensional projections and rotations are quite general and can easily be extended to even higher dimensions" (Noll, 1967b).

A number of programming languages have been especially designed for the artist and film maker. Kenneth Knowlton and his associates at Bell Laboratories, Murray Hill, N. J., have developed three such languages: Beflix, Tarps, and Explor.

Beflix, a film-making language in Fortran IV, provides drafting operations that draw rectangles, straight lines, arcs, other curves, and alphanumeric characters, as well as operations concerning the content of rectangular areas, the specification of the projection grid, operations related to elements labeled bugs (not to be confused with the undesirable bugs that occur in many computer programs), and miscellaneous operations having to do with output, debugging, etc. Tarps is a two-dimensional alphanumeric raster picture system for the production of designs, diagrams, and textures, usually for movies. Tarps contains operations such as ZOOM, which progressively enlarges the image by an integral factor; ESPEZOOM, which gives a clockwise (right) or counterclockwise (left) spiraling zoom; and SYM, which produces an enlarging kaleidoscope image.

Explor-a generator of images from explicit patterns, local operations, and randomness-is a system for computer generation of still or moving images from explicitly defined patterns, local operations, and randomness. Explor produces output images comprising rectangular arrays (240 by 340) of black, white, and "twinkling" dots. Among its

intended scientific and artistic applications are the "production of stimuli for visual experiments, the depiction of visual 'phosphors' such as moving checkerboards and stripes, and picture processing." This system may also be used to simulate a "variety of two-dimensional processes and mechanisms, such as crystal growth and etching, neural (e.g., retinal) nets, random walk, diffusion, and iterative arrays of logic modules."

Explor is a macrolanguage with sets of instructions for picture output, for changing the internal array, for defining patterns, for flow of control, and for instruction modification. It provides facilities for specifying periodic and/or random applications of its operations, and flexible means of specifying uniform or locality-dependent translation of internal symbols. Another approach to programming languages for the artist permits the artist to define his own vocabulary, built on an REL (rapidly extensible language) support system, as he creates his art (Thompson et al., 1969).

The use of the computer for sculpture is ordinarily to use graphic facilities, such as those already described for painting and film making, to display a range of design possibilities and perspectives for proposed sculpture. Programs that produce paper tapes, which in turn are used to drive milling machines that actually produce artifacts, have also been designed (Mallory, 1969).

Music. Computer-generated music produces either scores, which are then performed, or music for which the acoustic properties are specified and which is then synthesized by a digital-to-analog converter.

Musicomp is a compositional programming language that provides techniques for generating original musical scores as well as for synthesizing music (Hiller, 1969). An adaptable language that can be changed and expanded according to need, Musicomp consists of three basic parts: system regulatory routines, compositional and analytical subroutines, and sound synthesis routines. The latter "are not actual synthesis programs but rather routines that prepare and organize data that serve as input to sound-generating programs." Among Musicomp's compositional subroutines are a subroutine for choice of a rhythmic mode; a subroutine that provides choice of a range of pitches and then of a specific pitch from within this range; a subroutine that controls the melodic range rule (in a single line a limit such as an octave is imposed on melodic motion); and others. The routines provide for the generation of phrases and their imitations and per-

ARTS APPLICATIONS

mutations; generation of all the permutations of the given role of n items; generation of similar rhythmic data from more than one instrument for any length of time; and generation of dynamics indications in playing styles according to serial processes. Musicomp can be used in conjunction with Fortran.

Output of musical scores either uses standard printer conventions that can then be translated onto regular music staves, a cathode-ray tube configuration such as that described earlier in terms of input of musical scores, or a plotter to draw a score. In one such program written for a Calcomp plotter, the notation makes use of a select set of symbols already available in the Calcomp library (triangles, circles, lines, and so on). The "language of the scores is given by the distribution, size, and position of symbols on each page of the scores; in effect, each page is a plot of dynamic level versus time. Once the performer is provided with a short introductory explanation of the notation, he is able to perform directly from the score" (Hiller, 1970).

Computer-generated scores have ranged from music that is highly random in character to music that is intended to approximate some aspects of the style of a given composer or musical genre. In fact, in the course of some experiments, a range from greater to less randomness is apparent (Moorer, 1972). Analysis of church hymns (Brooks et al., 1957), using Markovian analysis, resulted in simulated hymns with considerable randomness when low-order probabilities (probabilities of orders 1 through 8 were assigned in this experiment) were used. The use of high-order probabilities produced parts of original hymns connected together. The *Illiac* Suite, one of the best known early computer compositions, was a string quartet comprising rhythms, chosen at random, and melodies constructed according to some rules of classical harmony and counterpoint. Another computer-generated composition, *Sonoriferous* Loops, written for instrumental ensemble and tape, used four parameters: pitch, register, rhythmic unit, and rhythmic mode. Rhythmic modes (basic metrical patterns) and rhythmic units within those modes were chosen in accord with assigned probability distributions; probability distributions that differed for each instrument and for different parts of the composition were also used to make choices between rest or play and for octave registers (Hiller, 1970). A recent effort to generate scores for western popular music (Moorer, 1972) proceeded according to the following sequence:

the overall form of the piece is chosen first,
the chords are chosen second, and the melody,

last. . . the choice of chords before melody is to avoid the problem of deciding what chords should go with a given melody. The problem is reversed and simplified to constraining some number of the prospective melody notes to lie in the chord. The overall structure is similarly chosen first, to prevent attempting to derive the structure from the melody. . . . The overall structure is decided upon by first choosing two numbers, the number of major groups and the number of minor groups. Each of these numbers is constrained to be a power of two times the number of beats per measure, a parameter supplied by the operator. The total piece length is then the product of the number of major groups times the number of minor groups times the number of beats per minor group.

Two parameters implicit in some of the choices made in experiments with computer-generated music are the periodicity of the melodic line and the consonance and/or dissonance implied by the structures of chords. Periodicity implies the repetition and transformation of melodic groups. Consonance and dissonance are subjective matters, but some combinations of notes such as a perfect fifth or a major third are considered to be more consonant than others (Moorer, 1972). Another statement on consonance and dissonance notes that "tones are most dissonant when their frequencies are separated by a particular fraction of a critical bandwidth and are consonant when their frequencies coincide or are separated by more than a critical bandwidth" (Pierce and Mathews, 1969). This statement continues by noting that partials "not crowded too close together at high frequencies" can be pleasing or consonant.

Composers who use the computer to synthesize sound rather than to produce musical scores are concerned with many of the same elements, although they are parameterized according to acoustic categories rather than graphic symbols. The composer using sound synthesis must be concerned with amplitudes, frequencies, amplitude modulation rate, and waveform specification. According to one set of specifications, the digital-analog converters used in music synthesis must be capable of converting 12 to 14 bit words at a rate of 40,000 per second. "The long word length ensures adequate dynamic range and the high rate, a sufficiently broad frequency response" (Freedman, 1969). Additionally, for pleasant sound, the computer equipment must have adequate block transfer rates between auxiliary storage and main memory so that there will be continuous reproduction of sound without breaks.

Because electronically synthesized sound lacks some of the variability to which the human ear is accustomed, composers who are generating sound electronically have concerns with which those who are generating scores are not troubled. For example, electronically sustained tones seem to provide a constant stimulus that may fatigue the ear. To achieve a more pleasing timbre, it has been suggested (Roberts, 1969) that any one of the following techniques might be used: "(1) control of the amplitude envelope: crescendo, diminuendo, exponential decay, and so on; (2) variation of the pitch-vibrato or glissando; (3) variation of wave form." Roberts also notes that "another feature of electronic tone that the listener finds distressing, or at least unnatural, is that quality is independent of loudness. . . . It is, of course, easy to correct this feature by introducing deliberate nonlinearities into the computer-simulated oscillators."

As is the case with computer-generated art, the field of computer-generated music is very extensive and cannot be done full justice in an article of this length. For a useful survey of music composed with the computer, the reader is referred to Hiller (1970).

Dance. Computer-generated dance is in a much more embryonic state. One experiment used the computer to specify the number of dancers on the stage at any given time and to indicate the positions they would occupy on that stage throughout the period of time they were on stage (Sagasti and Page, 1970). Another experiment (Noll, 1967a) used a computer display to delineate the spatial and arm movements of a group of dancers on stage. The difficulties encountered in pattern analysis for the dance also obtain for pattern generation. That is, specifying the intricacies of movements entailed by dance is exceedingly complicated and, as yet, beyond the reach of any current experiment.

REFERENCES

1957. Brooks, F. P., A. L. Hopkins, P. G. Newmann, and W. V., Wright. "An Experiment in Musical Composition," *IRE Trans. Electronic Computers*, EC-Vol. 6, No. 1, September, pp. 175-182.
1964. Brook, Barry, and Murray Gould. *Notating Music with Ordinary Typewriter Characters: A Plaine and Easie Code System for Musicke*. Flushing, N. Y.: Queens College of the City University of New York.
1967. ———. "Music Bibliography and the Computer." In Gerald Lefkoff (Ed.), *Computer Applications in Music*. Morgantown: West Virginia University Library, pp. 9-27.
- 1967a. Noll, Michael. "Choreography and Computers," *Dance Magazine*, January.
- 1967b. ———. "Computers and the Visual Arts," *Design and Planning*, No. 2, Hastings House Publications, Inc.
- 1967c. ———. "The Digital Computer as a Creative Medium," *IEEE Spectrum*, Vol. 4, No. 10, October, pp. 89-95.
1968. Csur, Charles, and James Shaffer. "Art, Computers and Mathematics," *A FIPS-Conference Proceedings*, Vol. 33. Wayne, Pa.: MDI Publications, pp. 1293-1298.
1968. Guzman, Adolfo. "Decomposition of a Visual Scene into Three-Dimensional Bodies," *A FIPS-Conference Proceedings*, Vol. 33, pt. 1. Wayne, Pa.: MDI Publications, pp. 291-304.
1969. Beauchamp, James W. "A Computer System for Time-Variant Harmonic Analysis and Synthesis of Musical Tones." In Heinz von Foerster and James W. Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 19-62.
1969. Freedman, M. David. "On-Line Generation of Sound." In Heinz von Foerster and James Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 13-18.
1969. Hiller, Lejaren. "Some Compositional Techniques Involving the Use of Computers." In Heinz von Foerster and James W. Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 71-83.
1969. Mallery, Robert. "Computer Sculpture: Six Levels of Cybernetics," *Art Forum*, May, pp. 29-35.
1969. Mathews, M. B., and L. Rosler. "Graphical Language for the Scores of Computer-Generated Sounds." In Heinz von Foerster and James W. Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 841-14.
1969. Pierce, J. R., and M. V. Mathews. "Control of Consonance and Dissonance with Nonharmonic Overtones." In Heinz von Foerster and James W. Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 63-68.
1969. Roberts, Arthur. "Some New Developments in Computer-Generated Music." In Heinz von Foerster and James W. Beauchamp (Eds.), *Music by Computers*. New York: John Wiley, pp. 63-68.
- 1969a. Rosenfeld, Azriel. *Picture Processing by Computer*. New York: Academic Press.
- 1969b. ———. "Picture Processing by Computer," *Computing Surveys*, Vol. 1, No. 3, September, pp. 146-176.

ASCII

1969. Thompson, F. B., P. C. Lockeman, B. H. Dostert, and R. S. Deverill. "REL: A Rapidly Extensible Language System," *Proceedings 24th National ACM Conference*, August, Vol. 24.
1970. Canaday, John. "Less Art, More Computer, Please." *The New York Times*, Aug. 30, Section D, p. 19.
1970. Fiore, Mary E. "Webern's Use of Motive in the Piano Variations." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 115-122.
1970. Fuller, Ramon. "Toward a Theory of Weberian Harmony, Via Analysis with a Digital Computer." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 123-131.
1970. Hiller, Lejaren. "Music Composed with Computers-A Historical Survey?" In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 49-96.
1970. Hudson, Barton. "Toward a Comprehensive French Chanson Catalogue." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 277-287.
1970. Jackson, Roland. "Harmony Before and After 1910: A Computer Comparison." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 132-146.
1970. Lieberman, Fredric. "Computer-Aided Analysis of Japanese Music." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 181-192.
1970. Sagasti, Francisco, and William Page. "Computer Choreography: An Experiment on the Interaction Between Dance and the Computer," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 3, No. 1, January, pp. 46-49.
1970. Sedelow, Sally Yeates. "The Computer in the Humanities and Fine Arts," *Computing Surveys*, Vol. 2, No. 2, June, pp. 89-110.
1970. Suchoff, Benjamin. "Computer-Oriented Comparative Musicology," In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 193-206.
1970. Youngblood, Joseph. "Root Progressions and Composer Identification." In H. B. Lincoln (Ed.), *The Computer and Music*. Ithaca, N.Y.: Cornell University Press, pp. 172-180.
1971. Cantor, Dawn. "A Computer Program That Accepts Common Musical Notation," *Computers and the Humanities*, Vol. 6, No. 2, November, pp. 103-109.
1971. Whitney, John H. "A Computer Art for the Video Picture Wall," *Proceedings: International Federation for Information Processing*, August.

1972. Chenhall, Robert G., et al. *Report of Museum Data Bank Study Group*. Hershey, Pa., March 27-28. (Available from Dr. Chenhall, Margaret Woodbury Strong Museum, Rochester, N.Y.)
1972. Moorer, James Anderson. "Music and Computer Composition." *Communications of the ACM*, Vol. 15, No. 2, February, pp. 104-113.

S. A. SEDELOW,

ASCII

For articles on related subjects see **CODES**; and EBCDIC.

The American Standard Code for Information Interchange (ASCII) is a seven-bit code also known as the USA Standard Code for Information Interchange (USASCII).

Because eight-bit codes are much more common on computers than seven-bit codes, ASCII is commonly embedded in an eight-bit code, ASCII-8, which is shown as Exhibit 1. This arrangement shows the 128 ($=2^7$) possible combinations for the seven-bit USASCII code. The leftmost four bits (or first hexadecimal digit) of the eight-bit code are shown as column heads across the top and the rightmost four bits (or second hexadecimal digit) are listed on the side. Thus, for example, we have

Character	Code	
	Binary	Hexadecimal
4	01010100	5 4
Y	10111001	B 9
c	11100011	E 3
=	01011101	5 D

The meanings of the control characters and special graphic characters are shown below the illustration.

I. FLORES

ASSEMBLERS

For articles on related subjects see **INDEX REGISTER**; **LANGUAGE PROCESSORS**; **LINKAGE EDITOR**; **LOADER**; **MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING**; **MACROINSTRUCTION**; **OPERAND**; and **OPERATION CODE**.

For articles on related terms see **EDSAC**; **IBM 360-370 SERIES**; and **READ-ONLY STORE**.

Bit Positions 4, 3, 2, 1	Second Hexadecimal Digit	00				01				10				11			
		00	01	10	11	00	01	10	11	00	01	10	11	00	01	10	11
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL	DLE			SP	0					@	P			v	p
0001	1	SOH	DC1			!	1					A	Q			a	q
0010	2	STX	DC2			"	2					B	R			b	r
0011	3	ETX	DC3			#	3					C	S			c	s
0100	4	EOT	DC4			\$	4					D	T			d	t
0101	5	ENQ	NAK			%	5					E	U			e	u
0110	6	ACK	SYN			&	6					F	V			f	v
0111	7	BEL	ETB			'	7					G	W			g	w
1000	8	BS	CAN			(8					H	X			h	x
1001	9	HT	EM)	9					I	Y			i	y
1010	A	LF	SUB			*						J	Z			j	z
1011	B	VT	ESC			+	;					K	[k	{
1100	C	FF	FS			,	<					L	\			l	!
1101	D	CR	GS			=						M]			m	~
1110	E	SO	RS			>						N	^			n	
1111	F	SI	us			/	?					O				o	DEL

Control Character Representations

NUL	Nu	DLE	Data Link Escape (CC)
SOH	Start of Heading (CC)	DC1	Device Control 1
STX	Start of Text (CC)	DC2	Device Control 2
ETX	End of Text (CC)	DC3	Device Control 3
EOT	End of Transmission (CC)	DC4	Device Control 4
ENQ	Enquiry (CC)	NAK	Negative Acknowledge (CC)
ACK	Acknowledge (CC)	SYN	Synchronous Idle (CC)
BEL	Bell	ETB	End of Transmission Block (CC)
BS	Backspace (FE)	CAN	Cancel
HT	Horizontal Tabulation (FE)	EM	End of Medium
LF	Line Feed (FE)	SUB	Substitute
VT	Vertical Tabulation (FE)	ESC	Escape
FF	Form Feed (FE)	FS	File Separator (IS)
CR	Carriage Return (FE)	GS	Group Separator (IS)
SO	Shift Out	RS	Record Separator (IS)
SI	Shift In	us	Unit Separator (IS)
		DEL	Delete
(CC)	Communication Control		
(FE)	Format Effector		
(IS)	Information Separator		

Special Graphic Characters

SP	Space	<	Less Than
!	Exclamation Point	=	Equals
	Logical OR	>	Greater Than
"	Quotation Marks	3	Question Mark
#	Number Sign	@	Commercial At
4	Dollar Sign	{	Opening Bracket
%	Percent	\	Reverse Slant
&	Ampersand	}	Closing Bracket
'	Apostrophe	^	Circumflex
(Opening Parenthesis	~	Logical NOT
)	Closing Parenthesis	~	Underline
*	Asterisk	~	Grave Accent
+	Plus	}	Opening Brace
,	Comma		Vertical Line (This graphic is stylized to distinguish it from Logical OR)
-	Hyphen (Minus)	}	Closing Brace
.	Period (Decimal Point)		Tilde
/	Slant		
:	Colon		
;	Semicolon		

Exhibit 1 ASCII-8

An assembler (contraction for assembly program) is a program that facilitates the preparation of programs at the machine language level by taking symbolic representations of individual (instruction or data) words and converting them into a form (binary or byte) suitable for input to a linker or loader. It permits the use of mnemonic names for function codes, allows symbolic names to be assigned to storage locations, provides facilities for address arithmetic in terms of such symbolic names, and (usually) enables the user to write numerical constants to various bases in a variety of forms.

An alternative, more abstract definition is as follows: An **assembly language** is a programming language in which the basic set of operations includes the operation codes of the machine, and whose data structure maps directly onto the store and registers of the machine. An assembler is a compiler for an assembly language.

Either of these definitions leads to the informal description often used, namely, that an assembler is a way of writing symbolic programs that allows the programmer full access to all the facilities of the real machine. (By contrast, higher-level languages such

ASSEMBLERS

as Fortran and PL/I provide an abstract machine that conceals some of the facilities available in the real machine.)

History. The term “assembly subroutine” for a routine that assembles a master routine and a number of subroutines into a single program was used in the first book about programming digital computers (Wilkes, Wheeler, and Gill, 1951) in connection with the EDSAC computer. Indeed, EDSAC had a rudimentary assembler called Initial Orders, which allowed the user to write machine instructions consisting of a single alphabetic-letter instruction code, a decimal address, and a terminating letter, which caused one of 12 preset quantities to be added to the address at assembly time. (The Initial Orders were implemented in a kind of read-only store consisting of a wired telephone uniselector.)

Although the use of a symbolic representation of machine language programs now seems so obviously desirable as not to be worth discussing, this was not always so. A dichotomy of view existed right from the start in England, with the EDSAC group advocating a measure of symbolic programming, which was derided by the binary faction at Manchester (MARK I). This controversy is exemplified by the following quotation from Wilkes (1956),

... the utility or otherwise of elaborate conversion schemes is at present a matter of some controversy, ... there are ... people who will have nothing to do with any form of conversion. They believe that the programmer does best to write his orders in a form as near as possible to that which they will take inside the machine.

Probably the first assembler in the sense used in this article was SOAP (Symbolic Optimizer and Assembly Program) on the IBM 650 computer in the mid- 1950s. However, the symbolic assembly features of SOAP were not its main feature (the 650 was a decimal computer anyway, which removed some of the difficulties of direct machine language coding). The 650 had a magnetic drum memory and an instruction code in which each instruction specified the address of its successor. For maximum efficiency, instructions had to be placed on the drum in positions such that the execution of each instruction overlapped as far as possible the time for the drum to rotate to the next instruction position, thus minimizing the latency time waiting for instructions. Such minimum-access coding involved a very difficult optimizing process, and it was this that SOAP achieved.

The most significant event in the history of assemblers was the Symbolic Assembly Program (SAP) for the IBM 704. The original SAP assembler (UASAP) was written by programmers at United Aircraft Corporation and was distributed by the SHARE organization. SAP set the external form of an assembly language that was to be a model for all its successors, and which persists almost unchanged to the present day. On later versions of the 700 series computers, SAP was replaced by FAP (Fortran Assembly Program).

Facilities. A typical machine instruction consists of an operation code, an address, and one or more register fields. The address may refer to a data area or to another instruction (e.g., the destination of a transfer of control). A SAP-like assembler provides a fixed set of (usually mnemonic) function codes and an open-ended set of programmer-defined symbols for use in address parts. Such address symbols may be defined explicitly or (in the case of jump destinations) implicitly by attaching them as labels to particular instructions. Although a symbol stands for an address, the assembler cannot convert label symbols directly into addresses, since the address in storage into which a particular instruction will be loaded is not known at assembly time. (It is finally determined only when a number of routines are combined together to form a complete program.) The difficulty is resolved by recording as the value of the label symbol the displacement of the instruction in question from the beginning of the code for the subroutine, and marking it in the assembler output as a relative or relocatable value, to be adjusted later by the linker or loader.

Thus, SAP introduced the basic structure of a symbolic instruction as being made up of three fields:

1. *Location* (possibly blank). A symbol placed here takes as its value the address of the register in which the corresponding instruction will be stored: Thus it serves as a label by which the instruction can be referenced by other instructions.

2. *Operation* code. The symbol here is one of a fixed repertoire of operation-code symbols

3. *Operand*. This field is usually made up of a number of subfields, reflecting the address/register structure of the computer. The subfields may be simple integer constants or may be expressions made up of symbols (representing addresses), constants, and simple arithmetic operations (usually plus and minus). Alternatively, a literal operand may be supplied: The assembler will store this and substitute the appropriate address in the instruction.

The following fragment of SAP coding illustrates this structure:

```

      TRA ALPHA
      LOC 16385
ALPHA CLA BETA
      STO DELTA
SYMB FAD =3.14159
      SXO STMB-2,4
      STO SYMT

```

Each instruction in this example is made up of three fields: location (label), operation code, and address. The operation codes are mnemonic, e.g., **TRA** = transfer control, **CLA** = clear accumulator, **FAD** = floating add, etc. The address fields show the various possible constructions. In the first line the address is a symbol **ALPHA**, as yet undefined. (It appears as a label on a later instruction.) In line 2 the address is explicit, and in lines 3 and 4 symbols (presumably defined elsewhere in the program of which this fragment forms a part) are used. Line 5 illustrates the use of a literal operand: the "equals" indicates that the 3.14159 following is the actual value to be loaded by the **FAD**, not the address of the operand. The next line illustrates a more complex address: It is a two-field form in which the first component is a store address and the second component identifies an index register to be used; in this example the store address is specified as an expression.

The following excerpt of OS/370 assembler code for the IBM System/370 computers, whose purpose is to sum 13 numbers, shows how little things have changed in ten years:

```

      L    3, = F'0'  CLEAR REGISTER
      L    5, = F'0'  USING LITERAL
      LH   4, = H'14' LOAD REGISTER
      B    BCNT      ENTER LOOP
BNTER AH  5,STZ(3)  INDEX STZ BY REG 3
      AH   3, = H'2' INCREMENT INDEX
BCNT BCT 4,BNTER  BRANCH ON COUNT
*
      ST 5,BSUM
STZ  D C  H'15,225,1,52,10,48,76,42,88,26,14,4,32'
BSUM DC  F'0'

```

The three-field format is still used, though the mnemonics have changed. With the exception of the branch (**B**) order, which has a label as its address, the address field is made up of a register designator and a second field, which in these examples is either a

symbolic store address or a literal. (For certain instructions it might be another register designator.)

Literals are introduced by "equals," but now include a type code (**F** = full word, **H** = half-word). Indexing (modification) is illustrated in the line starting **BNTER**, and finally there are specifications of a number of constants introduced by the **DC** (Define Constant) pseudo-operation. The comments on the right are the part of the programmer's documentation of the program. To explain the program a bit more fully we note that the "14" in the third instruction is one more than the number of numbers to be added. The first **AH** instruction adds the contents of **STZ** plus register 3 to register 5, thus forming the sum. The next **AH** instruction increases the contents of register 3 by 2 so that the next number in the **STZ** list will be picked up by the previous instruction. The **BCT** instruction subtracts 1 from register 4 and transfers to **BNTER** as long as register 4 is positive. After 13 numbers have been added, register 4 becomes 0 and the sum is stored in **BSUM**.

Pseudo-Operations. Another important feature of assemblers, again first introduced in SAP, is the pseudo-operation. Its primary use is to convey information to the assembler about the way it is to deal with the program: In this respect it resembles the control combinations of the **EDSAC** Initial Orders. A secondary use of pseudo-instructions is to deal with constructions (e.g., numerical constants) that do not conform to the operation-code/address structure. The name "pseudo-operation" arises from the use of the operation-code field to specify the action that is to take place.

Important uses of pseudo-operations are in defining and manipulating symbols and in the allocation of storage. For example, the SAP pseudo-operation **BSS** reserves an area of storage and sets the address of the start of the area as the value of a symbol (so that the area can be referenced symbolically by other instructions). More precisely,

symbol **BSS** *integer-constant*

will reserve a block of storage of length *integer-constant* registers, and set the address of the first register as the value of *symbol*.

Other pseudo-operations allow explicit setting of symbol values; thus,

symbol **SET** *expression*

will set the symbol to the value given by the *expression* (which must consist of constants and/or

ASSEMBLERS

previously defined symbols), and

symbol-1 SYN symbol-2

will give *symbol-2* the same value as *symbol-1* (i.e., they become synonyms).

Another group of pseudo-operations is concerned with the cross-referencing between separately assembled pieces of code; thus

symbol COMMON n

defines the value of **symbol** as the address of a block of *n* registers in the **COMMON** area, and **ENTRY** is used to define the entry point(s) to a subroutine.

Numerical constants are also dealt with by pseudo-instructions. In **SAP**, pseudo-operations **OCT**, **DEC** were provided, the “operand” field for these consisting of a list of constants in octal or decimal notation, respectively, and the constants being converted into binary by the assembler. The pseudo-operation **BCD** allowed decimal constants to be converted to binary-coded decimal (packed decimal) format. In assembler code for the IBM 370, there is only a slight change: A single pseudo-operation **DC** (Define Constant) is followed by a list of constants, each of which is preceded by a format code to specify the type of conversion to be carried out.

Conditional assembly. A feature of many assemblers is the ability to selectively assemble pieces of program: This is particularly useful in package programs that have to provide a large number of options. In its simplest form this facility is provided by a pseudo-instruction that controls the assembly of the immediately following instruction. For example, there might be an operation **IFT** (if-true) such that

IFT symbol-1 relation symbol-2

will cause the next instruction to be assembled only if the relation between the symbols is true. The obvious converse **IFF** (if-false) will usually be provided also.

In more recent assemblers a more elaborate facility of assembly-time jumps and labels is provided. Typically, assembly-time labels (or sequence symbols) are preceded by a period and appear in the label field. However, they are ignored by the assembler except in the context of two new pseudo-instructions **AGO** and **AIF**. (The mnemonics are derived from “assembler **GOTO**” and “assembler **IF**”)

Let **.\$S** be a sequence symbol; then

AGO .SS

causes assembly to be continued from the line in which the symbol **.\$S** appears in the label field (usually, this must be a forward jump), and

AI F (symbol-1 relation symbol-2) .SS

causes assembly to be continued from the line labeled **.\$S** if the condition is true; otherwise, assembly continues with the next line of code, as usual.

Listings. An assembler usually provides a variety of information about the program that it has assembled. Besides details of any obvious errors such as incorrect syntax or multiple definition of symbols, the following may be provided:

1. Listing of symbolic instructions side by side with generated binary or binary-symbolic code.
2. Table of symbols defined in a routine, with or without their values.
3. Table of symbols used in a routine.
4. Cross-reference table: for each symbol defined, its name, value, and a list of all the instructions that reference it.

The form of the listing is generally controlled by one or more pseudo-operations; for example:

LIST FULL
LIST NONE
LIST SYMBOLS
etc.

Other **common** pseudo-operations are **EJECT**, which causes a page feed on the printer at the point in the listing where it **occurs**, and **SPACE n**, which causes a spacing of *n* blank lines in the listing. The listing corresponding to the program fragment for the IBM System/370 (given in the section “Facilities”) is shown in Fig. 1.

Macro Assemblers. An important attribute of an assembler is the ability to define and use macros. It often happens that a certain pattern of orders occurs in several places in a program with only minor variations. This is particularly the case if there is a common operation that requires several machine orders for its execution; for example, the calling sequence for a call of another routine. Thus,

LOC	OBJECT CODE	ADDR1	ADDR2	ST#	NAME	OP	OPERANDS
00000C	5830 2030		00038	8	L	3,	= F'O' CLEAR REGISTER
000010	5850 2030		00038	9	L	5,	= F'O' USING LITERAL
000014	4840 2034		00030	10	LH	4,	= H'14" LOAD REGISTER
000018	47F0 2010		00024	11	B	BCNT	ENTER LOOP
00001 C	4A53 2038		00040	12	BNTER	AH	5,STZ(3) INDEX STZ BY REG 3
000020	4A30 2036		0003B	13	AH	3,	= H'2' INCREMENT INDEX
000024	4640 2014		0001C	14	BCNT	BCT	4,BNTER BRANCH ON COUNT
000028	5050 2054		0005C	15	ST	5,BSUM	STORE SUM
				16	*		
000040	000F00E1 00010034	} (Hexadecimal equivalents of 13 numbers)		17	STZ	DC	H'15,225,1,52,10,48,76,42,88,26,14,4,32'
000048	000A00300046002A						
000050	0058001 A000E0004						
000058	0020						
00005A	0000 ← (Filler needed to align next instruction properly)						
00005C	00000000			18	BSUM	DC	F'O'

The **LOC** column shows the address of each instruction relative to the beginning of the program. The **OBJECT CODE** columns show the contents of the instructions as they will appear in memory. The **ADDR1** (not used in this example) and **ADDR2** columns give the effective addresses of the operands. Thus, assuming general register 2 holds the value 8, 2030 has the effective value $8 + 30 = 38$. The **ST#** column is a sequential line number for the programmer's convenience. Note that columns to left of **ST#** are all given in hexadecimal.

Fig. 1. Example of Listing from System/370 Assembler.

to call the routine **SUB** with parameters **A** and **B**, it might be necessary to write

```
LDX 4,*
TFR SUB
NOP A
NOP B
```

(The first instruction loads into register 4 the address of itself; i.e., its location in store. From this the subroutine can compute the return address to resume operation of the main program after the calling sequence. The parameters **A** and **B** are assumed to be addresses, and have been placed as the address parts of two no-operation [i.e., null] instructions.) Evidently, it would be convenient for the programmer to be able to write

```
CALL SUB,A,B
```

and have the calling sequence generated for him. The advantages of this approach are threefold. The programmer writes less; his program is more readable; and if at some future stage the calling sequence

is changed, a change at one place in the program will insure that all **CALLS** are changed without the need to alter each one individually.

(In SAP the **CALL** macro was built into the system, and was described as a pseudo-operation. This usage of the term "pseudo-operation" is no longer current.) A macro assembler allows the programmer to define macroinstructions as sequences of ordinary instructions, and provides a means of inserting variable information in the generated sequences.

The Working of the Assembler. The "classic" assembler takes a routine (or subprogram) and converts it into binary symbolic form for subsequent processing by a linkage editor. The conversion is accomplished in two passes (i.e., the source program is scanned twice). The basic strategy is very simple. The first pass through the source program collects all the symbol definitions into a symbol table, and the second pass converts the program to binary symbolic form, using the definitions collected in the first pass. (The merit of a separate pass to collect symbol definitions is that it

ASSEMBLERS

obviates the tricky situation that arises if an occurrence of a symbol has to be processed before the symbol has been defined.)

Although the program is scanned twice, only in the crudest systems is the physical source material read twice. If the assembler is reading directly from cards, then the source material can be copied onto magnetic tape or disk during the first pass, and so preserved for the second pass. In the environment of modern operating systems, the assembler will in any case read card images from tape or disk on the first pass. A good assembler may encode the information read in the first pass into a form that allows a more efficient second pass.

During the second pass, the assembler will have to recognize three sorts of quantities: absolute quantities, relocatable quantities, and references to externally defined symbols. In the simplest case, all relocatable quantities are expressed relative to an origin at the beginning of the routine. The assembler therefore has to categorize the symbols as it builds up the symbol table, and then check for illegal combinations in expressions. (For example, it is meaningless to add two relocatable symbols, though their difference may be a respectable absolute quantity.) The exact form of the output from the assembler depends on the linkage editor. Typically, the assembler might produce the following output:

Header	Name of routine,
RLB	Relocatable binary section: Consists of binary symbolic code and relocation information,
Definition table	Definitions of global symbols defined in the routine (i.e., symbols that will be referenced in other routines).
Use table	Details of use of global and common symbols in the routine (i.e., symbols used here but defined elsewhere).

The *definition table* carries information about symbols defined in this routine which are to have a global meaning. Since these may be absolute or relative, the table must carry this information as well as the value. In the case of a relative symbol the value is relative to the beginning of the routine.

The *use table* is more complex, since it records all occurrences of global symbols within the routine. Its exact form will depend on the facilities provided

by the assembler-in particular the circumstances in which global symbols can be used. (If multiplication of global symbols is allowed, an additional table, the *product use table*, is also required.)

If multiple location counters are used, an extra block must be output giving the amount of space used by the routine relative to each location counter. Each relocatable item will carry with it an indication of the relevant location counter.

Meta Assemblers. Assemblers for different machines have much in common. They organize symbol tables, evaluate expressions, and generate binary words from a number of symbolic fields. The idea of a *meta* assembler is to provide a system with these general capabilities, together with a means of describing (in machine-independent form) the assembly rules for a particular machine. The *meta* assembler accepts this description and then functions apparently as a normal assembler.

The idea of a *meta* assembler originated with Ferguson (1966), who produced the only published paper on the subject. The ideas described in the paper were utilized in the Metasymbol assembler for the SDS 900 series, in the Sleuth 11 and Utmost assemblers for the Univac 1107/8, and in the *Meta* system for the CDC 3300. An important feature of these systems (which is usually glossed over in their descriptions) is that the syntax of the input to a *meta* assembler is fixed. The meaning of the symbolic information can be defined by the user, but he cannot change the syntax. Thus, although it is possible in using a *meta* assembler to write an assembler for most machines, it is not possible to mimic an existing assembler. (This is one of the many differences between a *meta* assembler and a compiler-compiler.)

The essentially new features of a *meta* assembler are (1) the provision of compile-time procedures and functions, and (2) a mechanism whereby the programmer can define binary output formats and cause such binary output to be generated.

Superficially, the input to a *meta* assembler looks like input to any assembler; each line has three fields-label, operation, and operand. The label is optional: If there is a symbol in this field, it is assigned a value equal to the current location-counter value. The operation may be the name of a built-in system operation, in which case it is no different from a pseudo-operation in a conventional assembler. If the operation is not the name of a built-in operation, it is assumed to be the name of a programmer-defined procedure, which will be obeyed, taking the operand field as an argument.

This procedure may have the effect of generating some code, or may just perform housekeeping operations such as entering items in a table. It should be particularly noted that the procedure is obeyed during assembly. It is in many ways comparable to a macro, but instead of textual substitution we obey a piece of program *written in assembly language*. This may itself contain calls to other procedures.

The operand field contains an expression, or group of expressions, made up of symbols and/or constants. (A group of expressions separated by commas is called a set; these can be nested in the operand field.) These expressions are evaluated by the system in the same way that a normal assembler evaluates its address field. Unlike a normal assembler, the expressions may contain calls to user-defined functions.

Included in the built-in procedures are `GEN` and `GENB`, which output the values of the operand set as a sequence of words or bytes, respectively, and `FORM`, which allows the user to define a named template for binary output. Thus,

```
INSTR FORM 6,3,15
```

defines (for a 24-bit word machine) a template made up of 3 fields consisting of 6, 3, and 15 bits, and attaches the name `INSTR` to this template (`FORM` is a built-in operation that generates named templates for later use). Suppose that subsequent to the definition of `INSTR`, we write

```
INSTR LDA, 7, ALPHA+ 1
```

(Here `INSTR` is in the operation field, and the operand field is a set of three expressions.) This will cause the three elements of the operand set to be evaluated, truncated, and concatenated to form a 24-bit binary output word. (Note that this technique would allow the operation code of an instruction to be written as an expression!)

More elaborate constructions allow the operation code mnemonics of a conventional assembler to be defined as the names of multiple-entry points to a procedure using `FORM` to generate instruction words. In this way, and using procedures to produce the required effect for pseudo-instructions, a "conventional" assembler image can be built up.

"Higher-Level" Assemblers. At this point we should perhaps ask the question, "why are assemblers still used?" Higher-level languages are adequate for many problems, but one needs to resort

to assembly language if it is desired to have a close control over storage allocation and to have direct control over the machine's internal registers. Thus, assembly language finds its main use in the writing of operating systems and similar system software. In order to have direct control at the internal register level, an assembly language program is necessarily written at a fine level of detail, with each instruction representing a single primitive operation. An unfortunate effect of working at this level of detail is that programs are rarely as perspicuous as programs written in a higher-level language can be; it is impossible to write an assembly language program that displays clearly the structure of the underlying algorithm.

Recently there has been a development in the direction of "higher-level" or "Algol-like" assembly languages that attempt to combine fine control over machine registers and store with a structure that reflects the overall structure of the program; for example, repetition loops, conditional statements, and functions and procedures. The facilities provided in such a language must correspond fairly closely to the actual hardware. For example, we cannot include anything that depends on dynamic storage allocation if the underlying hardware does not provide such facilities. (Put another way, the compiler for an Algol-like assembly language cannot assume the existence of a "run-time system". Every source statement except a procedure call must compile into open code.) The precise facilities provided in a system will depend on the particular machine, but will typically include the following:

1. Symbolic names (identifiers) with associated types. The types will correspond to the storage units manipulated by the machine instructions; for example, on the IBM System/370 they would include byte, short integer, integer, real, and long real.
2. Reserved identifiers for machine registers. A synonym facility may also be provided to associate other names with registers.
3. Block structure, giving scopes to identifiers.
4. Conditional and compound statements.
5. One-dimensional arrays, but not multidimensional arrays (these cannot be accessed by simple indexing on most machines).
6. Procedures and functions. (Usually only one parameter will be possible, and the calling mechanism will be the Fortran call-by-address. This corresponds to passing the parameter as an address in an accumulator or general-purpose register.)
7. Simple expressions (but nothing involving temporary storage; all operators are of equal precedence).

ASSEMBLY LANGUAGE

dence and evaluation is by a simple left-to-right scan).

8. Provision for including basic assembly language (e.g., for input operations),

The first higher-level assembler was the PL/360 system described by Wirth (1968) in a classic paper. As its name implies, it was designed for the IBM System 360 machines. An Algol-like assembler for a small machine has been developed at the National Physical Laboratory in the United Kingdom; called PL5 16, it is designed for the Honeywell DDP5 16.

REFERENCES

- 1951 Wilkes, M. V., D. J. Wheeler, and S. Gill. *The Preparation of Programs for an Electronic Digital Computer*. Cambridge, Mass.: Addison-Wesley.
1956. Wilkes, M. V. *Automatic Digital Computers*. London: Methuen & Co.
1966. Ferguson, D. E. "The Evolution of the Meta-Assembly Program," *Commun. Assoc. Comput. Mach.*, vol. 9, p. 190.
1968. Wirth, N. "PL/360, A Programming Language for the 360 Computers," *J. Assoc. Comput. Mach.*, vol. 15, p. 37.
1971. Flores, I. *Assemblers and BAL*. Englewood Cliffs, N.J.: Prentice-Hall.
1972. Barron, D. W. *Assemblers and Loaders*, 2d ed. New York: American-Elsevier, 1972.

D. W. BARRON

ASSEMBLY LANGUAGE. See **MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING.**

ASSOCIATION FOR COMPUTING MACHINERY (ACM)

For article on related subject see **AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES.**

Purpose. The Association for Computing Machinery is the largest scientific, educational, and

technical society of the computing community. Founded in 1947, the Association is dedicated to the development of information processing as a discipline, and to the responsible use of computers in an increasing diversity of applications.

The purposes of the Association are: (1) To advance the sciences and arts of information processing, including but not restricted to the study, design, development, construction, and application of modern machinery, computing techniques and appropriate languages for general information processing, storage, retrieval, and processing of data of all kinds and the automatic control and simulation of processes. (2) To promote the free interchange of information about the sciences and arts of information processing, both among specialists and among the public in the best scientific and professional tradition. (3) To develop and maintain the integrity and competence of individuals engaged in the practices of the sciences and arts of information processing.

How Established. ACM was founded at Columbia University on Sept. 15, 1947, as the Eastern Association for Computing Machinery. A constitution and bylaws were adopted in September 1949. ACM was incorporated in Delaware in December 1954. The following have held the office of ACM president:

J. H. Curtiss, 1947
John W. Mauchly, 1948-1950
Franz L. Alt, 1950-1952
Samuel B. Williams, 1952-1954
Alston S. Householder, 1954-1956
John W. Carr III, 1956-1958
Richard W. Hamming, 1958-1960
Harry D. Huskey, 1960-1962
Alan J. Perlis, 1962-1964
George E. Forsythe, 1964-1966
Anthony Oettinger, 1966-1968
Bernard A. Galler, 1968-1970
Walter M. Carlson, 1970-1972
Anthony Ralston, 1972-1974
Jean E. Sammet, 1974-

Organizational Structure. The Association is organized into 12 regions, 11 covering the United States and Canada, and one encompassing western Europe. Each region is represented in the Council of the ACM (the elected governing body) by a regional

ASSOCIATION FOR COMPUTING MACHINERY (ACM)

representative. With an additional six members-at-large and the ex officio members (president, past-president, vice-president, secretary, treasurer, chairman of the Publications Board, and chairman of the Board of Special Interest Groups and Committees), the full Council comprises 25 members.

Each geographic region is subdivided into local chapters and student chapters. Presently there are approximately 100 local chapters and 125 student chapters.

The four classes of ACM membership and their qualifications are:

MEMBER—must subscribe to the purposes of the Association and have attained professional stature by demonstrating intellectual competence and ethical conduct in the arts and sciences of information processing.

ASSOCIATE—must subscribe to the purposes of the Association, but is ineligible for Member status.

STUDENT—full-time registrant at an accredited educational institution.

INSTITUTIONAL—institutions that subscribe to the purposes of the Association.

Total membership is about 30,000. The headquarters of ACM are located at 1133 Avenue of the Americas, New York, New York 10036.

Technical Program. The major organizational units of ACM devoted to technical activities of its members are the Special Interest Groups (SIGs) and Special Interest Committees (SICs). The SIGs and SICs operate as semiautonomous bodies within ACM for the advancement of activities in the following subject areas: Automata and Computability Theory; Computer Architecture; Artificial Intelligence; Business Data Processing; Biomedical Computing; Computers and the Physically Handicapped; Computers and Society; Data Communications; Computer Systems Installation Management; Computer Personnel Research; Computer Science Education; Computer Uses in Education; Design Automation; File Description and Translation; Computer Graphics; Information Retrieval; Language Analysis and Studies in the Humanities; Mathematical Programming; Measurement and Evaluation; Microprogramming; Numerical Mathematics; Operating Systems; Programming Languages; Symbolic and Algebraic Manipulation; Digital Simulation; Social and Behavioral Science Computing; University Computing Centers.

The National Lectureship Series was instituted in 1961 to enrich chapter activities by providing acknowledged specialists in various aspects of computing and its application. In 1966 ACM established

the Turing Award, given annually to an individual selected for his contributions of a technical nature made to the computing community. The award carries an honorarium of \$1,000. The recipients to date have been: Alan J. Perlis, Maurice V. Wilkes, R. W. Hamming, Marvin Minsky, J. H. Wilkinson, John McCarthy, Edsger W. Dijkstra, Charles W. Bachman, Donald E. Knuth, and (jointly) Allen Newell and Herbert A. Simon.

In 1970 the Distinguished Service Award was instituted. Its recipients have been Franz Alt, J. Donald Madden, George E. Forsythe, William Atchison, and John W. Carr III. A Programming Systems and Language Paper Award was established in 1969. In 1971, in conjunction with the twenty-fifth anniversary of the invention of the modern digital computer, ACM established the Grace Murray Hopper Award, to be given annually to the outstanding young computer professional of the year as nominated by ACM. To qualify, candidates must have been 30 years or younger at the time the qualifying contribution was made. The first award was to Donald E. Knuth.

The ACM publishes five major periodicals: *Journal of the Association for Computing Machinery* (1954, a quarterly) is devoted to highly technical papers of lasting value reporting on research and advances in the computing sciences. *Communications of the ACM* (1958, a monthly) publishes technical papers and timely articles on topics of interest to the computing profession. *Computing Reviews* (1960, a monthly) comprehensively covers the literature on computing and its applications. (A *Bibliography of Computing Literature*, keyed to *Computing Reviews*, but including additional materials, is published annually.) *Computing Surveys* (1969, a quarterly) presents comprehensive survey coverage of the state-of-the-art in the various areas of computing science and business data processing. *Transactions on Mathematical Software* (1975, a quarterly) publishes theoretical and applied articles on mathematical software as well as algorithms for computers.

ACM holds an annual conference that stresses technical programs and publishes a proceedings of the papers presented. ACM sponsors the annual Computer Science Conference, which is devoted mainly to brief reports of current research. ACM is a founding member of the American Federation of Information Processing Societies and participates in the annual National Computer Conference. In addition, the Special Interest groups and other subunits sponsor numerous technical symposia and meetings in North America and Europe.

I. L. AUERBACH

ASSOCIATION FOR EDUCATIONAL DATA SYSTEMS (AEDS)

ASSOCIATION FOR EDUCATIONAL DATA SYSTEMS (AEDS)

For article on related subject see **AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES**.

The Association for Educational Data Systems (AEDS) is a private, not-for-profit educational corporation founded and incorporated in Florida in 1962 by a group of professional educators and technical specialists oriented in educational applications.

The purpose of AEDS is to provide a forum for the exchange of ideas and information about the relationship of modern technology to modern education. Its objectives are:

- To provide a national association for educational data systems and encourage and assist the establishment of associations for educational data systems.
- To provide for sharing and exchanging ideas, techniques, materials, and procedures for use in modern educational data processing.
- To promote general recognition of the vital professional role played by the educational data processing specialist in a modern school system and the high level of competence required for this role.
- To promote and encourage appropriate use of electronic data processing and computing equipment and techniques for the improvement of education.
- To cooperate with manufacturers, distributors, and operators of educational data processing equipment and supplies in establishing and maintaining proper technical standards, and in meeting new needs for specialized devices and systems.
- To encourage and advise concerning research relating to educational data processing.

The following have served as presidents of the Association:

Robert Gates, 1962-1963
John Caffrey, 1963-1964
Don D. Bushnell, 1964-1965
C. Taylor Whittier, 1965-1966
John W. Sullivan, 1966-1967
Ernest Anderson, 1967-1968
John W. Hamblen, 1968-1969

Ralph Van Dusseldorp, 1969-1970
L. Everett Yarbrough, 1970-1971
Sylvia Charp, 1971-1972
Russell E. Weitz, 1972-1973
James Augustine, Jr., 1973-1975
Thomas McConnell, 1975-

The headquarters of AEDS is located at 1201 Sixteenth Street, N. W., Washington, D. C. 20036.

AEDS conducts an annual meeting dedicated to general and technical sessions on educational data processing.

The following publications are included in the membership fee: *AEDS Monitor*—a monthly publication giving current information about educational data processing; and *AEDS Journal*—a quarterly publication containing technical information about the development and specific applications of educational data processing.

I. L. AUERBACH

ASSOCIATION FRANÇAISE POUR LA CYBERNETIQUE ECONOMIQUE ET TECHNIQUE (AFCET)

For article on related subject see **INTERNATIONAL FEDERATION OF INFORMATION PROCESSING**.

Purpose. AFCET endeavors to bring together French scientists, engineers, users, and manufacturers working and interested in data processing, automation, and operational research, as well as in measurement and applied mathematics.

How Established. The organization was established in 1969 as a result of the amalgamation of: AFIRO (Association Française d'Informatique et de Recherche Operationnelle), AFRA (Association Française de Regulation et d'Automatisme), and AFIC (Association Française d'Instrumentation et de Controle). The merger was due to the relation between measurement and instrumentation as well as the relation of operational research to automation and data processing. The presidents of AFCET since 1969 have been

R. Mercier, 1969
J. Csech, 1970-1971
F. Genuys, 1972-1973
L. Euilysse, 1974

ASSOCIATIVE LANGUAGES

Organizational Structure. AFCET has 4,000 members at present. In addition, about 300 industrial companies are registered with AFCET and pay a much higher contribution than do members. The Association has various independent sections, each headed by a chairman. These include: AT—Automatique; CI—Composants Instrumentation; IG—Informatique de Gestion; SI—Systemes et Machines Informatiques; MA—Mathematiques; RO—Recherche Operationnelle; and AP—Applications Scientifiques et Industrielles. Its headquarters is located at Universite Paris IX Dauphine, 75775 Paris Cedex 16, France.

Technical Program. The sections organize seminars regularly (for 150 to 300 persons) on current subjects. Every year, AFCET holds a congress on data processing or on other subjects closely connected with its activities. It also participates and organizes international symposia such as the Fifth IMEKO Congress, Versailles 1970; the First IFAC/IFIP Symposium on Traffic Control, Versailles 1970; and the Fifth IFAC World Congress on Automatic Control, Paris 1972.

Publications. The Association issues four official publications :

Rairo (*R e v u e* Française d'Automatique, d'Informatique et de Recherche Operationnelle), published, in four series of three issues each, per year. This review contains reports of a very high level on the work done in laboratories and research institutes, Ph. D. theses, etc.

Automatisme, published in ten issues per year; covers industrial applications of automatic control and computer systems.

Informatique de Gestion, published in ten issues during the year; treats administrative data processing systems and computers.

Mesures, published in ten issues over the year; covers measurement, instrumentation, and components.

I. L. AUERBACH

ASSOCIATIVE LANGUAGES

For articles on related subjects see **ARTIFICIAL INTELLIGENCE**; **INFORMATION AND DATA**; **INFORMATION RETRIEVAL**; **LIST PROCESSING LANGUAGES**; **LISTS** and **LIST PRO-**

CESSING; **PROGRAMMING LANGUAGES**; and **STRING PROCESSING LANGUAGES**.

For article on related term see **HASHING**.

Associative language research has been motivated by two considerations. The languages might be useful in associative computers or could simply be a better way of stating algorithms. Applications have been suggested or carried out in essentially all nonnumerical problems and in a few special numerical ones such as sparse matrix calculations.

The essential fact about an associative memory is that it does not rely on explicit addresses. A reference to information contained in a memory cell is specified by a partial description of its contents. All cells in the memory which meet the specification are referred to by the statement. A conventional coordinate-addressed memory can be thought of as a special case in the following way: Each cell of the associative memory will be a pair consisting of a conventional cell and its address:

Associative cell :

address	conventional cell
---------	-------------------

To access a cell in this special associative memory, one specifies the contents of one particular part (the address field) of a cell. In the general associative memory a cell can be accessed by specifying the contents of any part of the cell.

The following example may help point out the importance of this seemingly minor difference. Suppose one were to store the contents of a telephone directory in a computer memory. There are several ways in which the data could be organized so that the telephone number of a given person could be found fairly quickly. However, the problem of going from a telephone number to its owner is rather difficult. One could, of course, enter a second directory ordered by numbers rather than names, but this entails representing the same information twice in the memory. There are several other possibilities, but all lead to compromises between inefficiency in time and inefficiency in storage. In an associative memory either question could be answered in one memory access and without any redundant information being stored in the memory.

This example is so clear and striking that it should be suspect. First of all, it is often the case that one person has several telephone numbers or that several people are listed for the same number. This is the so-called multiple-hit situation and is at the root of many of the problems encountered in using associative memories. Many other difficulties arise in the design of hardware associative memories.

ASSOCIATIVE LANGUAGES

There is even a theoretical basis for questioning the inherent practicality of a hardware associative memory. However, the idea of referring to information by a partial specification of its contents is intriguing and has found its way into a number of programming systems.

Let us consider the behavior of an associative memory each of whose cells contains an ordered 3-tuple:

3-element associative cell:

a	o	v
-----	-----	-----

written $a \cdot o \equiv v$. The symbols a , o , and v can be thought of as mnemonics for attribute, object, and value.

The elements of the 3-tuple are drawn from a universe of items; a 3-tuple of items is an association. Typical associations might be

father . john doe \equiv don doe
end . line \equiv point

If a universe of associations were stored in an associative memory, any partial specification should lead to retrieval. Letting x , z represent unspecified positions in an association, Table 1 enumerates the partial specifications (forms) possible in a 3-element associative memory.

Alternatively, Table 1 can be viewed as a generalization of the property list features of languages such as Lisp and IPL-V, and of the records (structures) features of languages such as Algol 68, Simula, and PL/I. In all these systems there are provisions for directly treating forms FO and FI and handling F3 and F6 fairly efficiently. The associative languages treat the association symmetrically and attempt to process efficiently all the forms of Table 1.

The basic operations on the universe of associations are entry, removal, and retrieval. Typically,

MAKE father . tom \equiv bill

will place the new association in the universe if it is not already there. Similarly,

ERASE father . ANY \equiv bill

will erase all associations that match the specification.

The heart of any associative system is its retrieval capability. One problem is that a retrieval statement is, in general, multiple-valued. This has led to the inclusion of sets in several systems and to

Table 1

Form Name	Form	Example	Interpretation
F0	$a \cdot o \equiv v$	son . john \equiv don	The association itself, if in memory
F1	$a \cdot o \equiv x$	son . john $\equiv x$	Sons of john
F2	$a \cdot x \equiv v$	son . $x \equiv$ don	Father of don
F3	$x \cdot o \equiv v$	x . john \equiv don	Relation of john to don
F4	$a \cdot x \equiv z$	son . $x \equiv z$	All father-son pairs
F5	$x \cdot z \equiv v$	$x \cdot z \equiv$ don	All associations with don as third component
F6	$x \cdot o \equiv z$	x . john $\equiv z$	All associations with john as second component

statements like sons \leftarrow son . john, which assigns to *sons* all the sons of john.

Another technique is to have iteration statements ranging over all values; e.g.:

foreach x **suchthat** son . $x \equiv$ don **do** . . .

The situation becomes much more complicated when there are several unspecified elements in a retrieval request. For example, consider

foreach x, y, z **suchthat**
father . $x \equiv y$ and father . $y \bullet z \equiv z$ **do** (1)
make granddad . $x \equiv z$.

Statement (1) requires solving for three variables. It is clearly inadequate to solve for each variable **independently**; the system must form an assignment of items that satisfies the associative context. For example, if the associations of the universe were

father . tom \equiv bill
father . pete \equiv tom
father . bill \equiv don
father . george \equiv clyde

statement (1) would yield assignments

$(x, y, z) = (\text{tom}, \text{bill}, \text{don})$
 $(x, y, z) = (\text{pete}, \text{tom}, \text{bill})$

and after the execution of statement (1) the universe would also contain the associations

granddad . tom \equiv don
granddad . pete \equiv bill

Treating statements like this requires a system to make use of intermediate structures and a fairly sophisticated interpreter. It should be noted that *no associative processing hardware has been suggested which can directly solve the general case of the problem above*. Statement (1) also suggests yet another way to view associative languages-as a generalization of pattern matching systems (e.g., SNOBOL) to move complex structures. If we picture an association as a labeled arrow in a graph, e.g.,

john $\xrightarrow{\text{son}}$ don

the associative language is a graph-matching system.

Most associative languages incorporate the concepts described above, but differ in other ways. One natural extension is to drop the restriction to triples. Most of the research on n-ary relations has been done in work oriented toward information retrieval (Codd, 1970). Although there is no clear demarcation, the following points generally separate the two fields. Associative languages are designed to (1) be used by programmer, (2) respond in fractional seconds, (3) have their data base fit in secondary storage. Information retrieval presupposes very large files, naive users and a somewhat slower response. There is currently no known mechanism for efficiently handling all the forms analogous to Table 1 for n-ary relations.

One problem that is common to information retrieval and associative languages is the question of how much deduction to incorporate. For example, one probably wants a system to use the fact that "bigger" is transitive in retrieval. Tramp (Ash and Sibley, 1968), for example, incorporates a few simple deduction rules. The problem is where to stop; one could incorporate many such rules or even a general theorem prover. At this point, the associative retrieval problem becomes part of artificial intelligence. In fact, all the new AI languages (Sussman and McDermott, 1972) incorporate associative retrieval features, including mechanisms for retrieving implied relations.

There has been quite a bit of work on the problem of efficiently implementing associative languages. Most systems employ a combination of hash-coding (Feldman and Rovner, 1969) and list-processing techniques. This work overlaps similar

studies in data structures and information retrieval. A related problem is to retrieve compound requests (like statement 1) efficiently; one should not, for example, find all males and then select only those over 7 feet tall. Some work has been done along these lines, but much more could be done.

REFERENCES

1968. Ash, W. L., and E. H. Sibley. "TRAMP: An Interpretive Associative Processor with Deductive Capabilities," *Proc. ACM 23rd National Conference*, pp. 143-156.
1969. Feldman, J. A., and P. D. Rovner, "An ALGOL-Based Associative Language," *Communications of the ACM*, Vol. 12, No. 8, pp. 439-449.
1970. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM*, Vol. 13, No. 6, June, pp. 377-387.
1971. Findler, N. V., J. L. Pfaltz, and H. J. Bernstein+ *Four High-Level Extensions of FORTRAN IV, SLIP, AMPPL II, TREETRAN, and SYMBOLANG*, Part II. New York: Spartan Books.
1972. Sussman, G. J., and D. V. McDermott. "From PLANNER to CONNIVER-A Genetic Approach," *Proc. AFIPS Fall Joint Computer Conference*, Vol. 41, Part II, pp. 1171-1179.

J. FELDMAN

ASSOCIATIVE MEMORY

For articles on related subjects see **ADDRESSING**; **CACHE MEMORY**; and **MEMORY**: Main.

For article on related term see **ATLAS**.

Data in an associative memory (Lewin, 1972) or content-addressable memory is not accessed by address as in conventional memory. Rather than being identified by the name of its location, it is identified from properties of its own value. To retrieve a word from associative store, a search key (or descriptor) must be presented which represents particular values of all or some of the bits of the word. This key is compared in parallel with the corresponding lock or tag bits of all stored words, and all words matching this key are signaled to be available. If the key is loose, with few attributes, it will access many words. The memory might indicate

ASSOCIATIVE MEMORY

the number of such words and would in any case normally provide each of these in turn for examination. The order in which they are presented is usually related to their order in physical storage and tells nothing of their value. Once available, each word can be used or, if not wanted, flagged (by a change of a single search bit) so that succeeding words can be retrieved.

It is obvious that associative search can be fairly complex if the search key has few elements and the association is loose. A limiting case is that in which at most one occurrence of a search key match exists. This is the case in the use of associative stores between levels in a memory hierarchy where the associative store is a scratch pad notating the existence of a copy of a record in the next higher-level store. Such is also the use of associative storage

in cache memory management.

Various attempts have been made to build associative processors in which the main memory is partially or completely associative (Lewin, 1972). In this case each memory word, in addition to match or equality logic on each bit, has other facilities such as "greater than" detection.

Fig. 1 illustrates the logical structure of an associative store, showing the possibility of search inputs ("don't care") whose value results in a positive match independent of the bit stored in the associative memory. In practice, such a store may be implemented using magnetic core or integrated circuit technology. In addition to the associative search mechanism, the memory is usually equipped with conventional read/write facilities (not shown in Fig. 1).

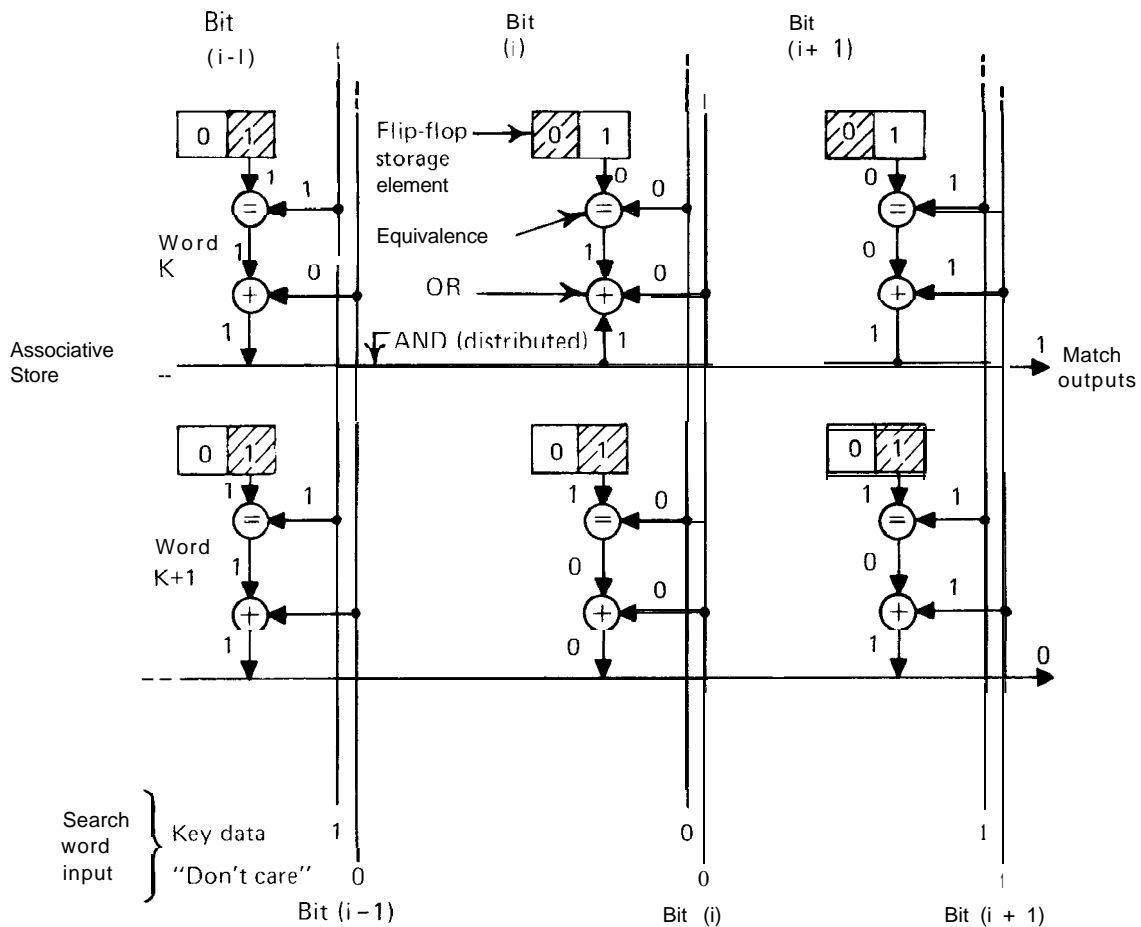


Fig. 1. Schematic of an associative store of two words of three bits each being accessed by match on two bits and by "don't care" on one.

ASSOCIATIVE MEMORY

A flip-flop is shown as the basic storage element and is shaded to indicate its current state. An equivalence circuit (denoted by =) is used to make the comparison with the search key. An additional OR circuit (denoted by +) per bit allows the use of a "don't care" search condition. The output from each bit is combined in an AND, represented by the horizontal line on the figure. A match output is 1 if and only if the stored data and key data bits match everywhere that "don't care" inputs are zero. Fig. 1 shows a match between a stored value of (1,0,0) and a search key of (1,0,X), where X indicates the "don't care" condition.

One important application of associative storage is in paging memory management. In such systems, of which the Atlas computer was an early major example, a relatively small, fast, main store is used in conjunction with a slow, large bulk or backup store. Each memory is divided into blocks or pages whose size may vary typically from 64 to 1,024

words. The small main memory will hold a small number of these, say 8 to 32, while the bulk store may have a capacity of thousands of pages. Data is transferred between main and backup store in **page-size** quantities. When the central processor requires a new word of information, high-order bits of its address are checked against appropriate bits of an associative page-address store in order to ascertain the existence in main memory of the word sought. If the page is already present, the associative store provides additional bits, giving its location in main store. If not present, related central processing unit (CPU) activity is halted while the page-memory access mechanism establishes a main store page location whose contents are returned to backup storage, after which this area of main store is refilled with the required new page.

A paging organization utilizing an associative page lookup is shown in Fig. 2. In this example a backup store of 1,024 pages of 64 words each is

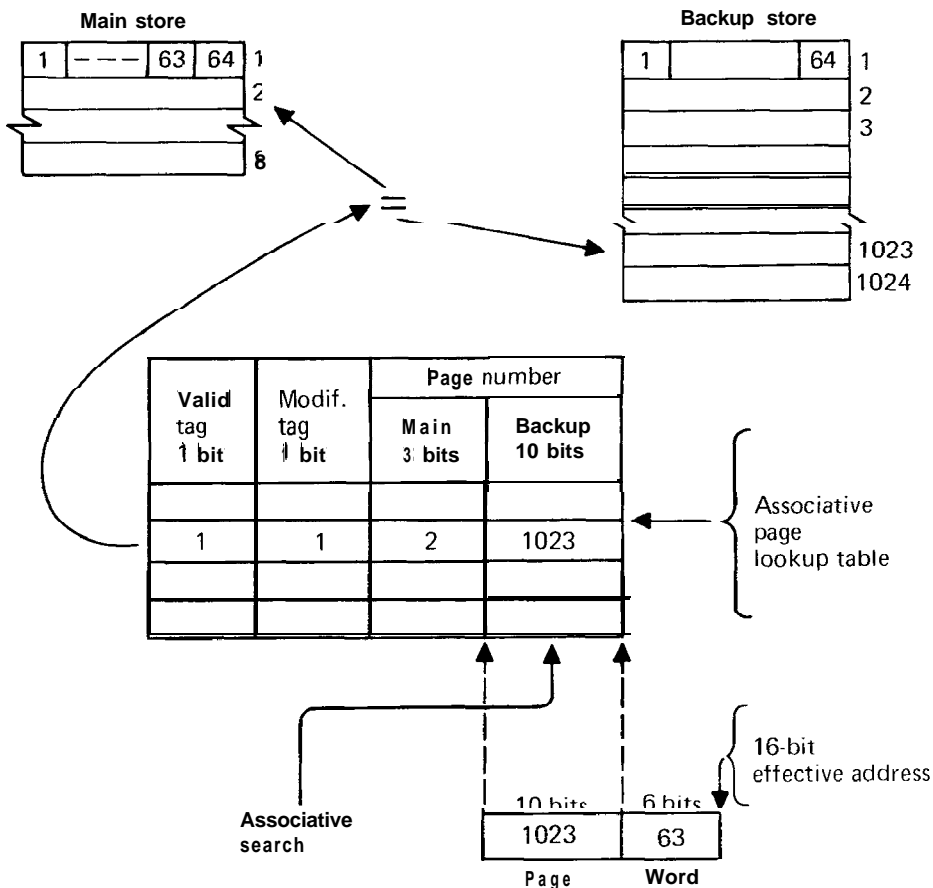


Fig. 2. A paging organization using an associative lookup table.

ATANASOFF, JOHN VINCENT

illustrated. The main store has a capacity of 8 pages of 64 words. An associative page lookup table of 4 words of 15 bits each is used to maintain order. In use, the upper 10 bits or page address of the effective address (1023) from the CPU is compared associatively with ten-bit page number in the associative store. If a match occurs, 5 bits are provided (3, 1, 1), giving the page location in main store (2) as well as two data integrity tags (1, 1). The valid tag indicates that the data in main store is a valid copy of that in backup store. The modified tag indicates that the data in main store has been modified by the CPU and is therefore not the same as in backup storage.

The result in the preceding example is that the word sought, number 63 from page 1023, is present at page 2, word 63 of the main store, that it is valid, and may differ from its image in backup store. Had the desired page not been in main store, one page of main store would have been selected for replacement, ideally one whose valid tag was zero, indicating it to be no longer needed. If all valid tags were 1, a page would be selected for replacement whose recent use is low (as established, for example, by automatic decrementing, at fixed time intervals, of a use counter associated with each page in the associative page-lookup table). This page, if its modified tag is 1, is stored in backup store and then replaced in main storage by the newly required page. The lookup table is modified accordingly to reflect a new backup page number for the corresponding main store page with validity tag 1, and modified tag 0.

REFERENCE

1972. Lewin, D. *Theory and Design of Digital Computers*. Camden, N. J.: Thomas Nelson, chaps. 6 and 9.

K. C. SMITH AND A. S. SEDRA

ATANASOFF, JOHN VINCENT

For articles on related subjects see **DIGITAL COMPUTERS**: Early; **ECKERT, J. PRESER**; and **MAUCHLY, JOHN W.**

John Atanasoff (b. Hamilton, N.Y., Oct. 4, 1903) received his B.S. at Florida State in 1925 and his M.S. at Iowa State in 1926.

Impeded by the cumbersome solving of large systems of equations and other complex calculations while working on his Ph.D. (Wisconsin, 1930, in math-physics), Atanasoff dismantled desk calculators in an attempt to adapt them and increase their computational capabilities. After receiving his doctorate, he returned to Iowa State in 1930, where he remained until 1945 as a professor in mathematics and (later) in physics. During his tenure there, he revamped an IBM punched card machine in another attempt to speed tedious calculations for his graduate students.



Fig. 1. John Vincent Atanasoff

Rejecting analog devices because they were too slow and not accurate enough, he concentrated on the digital approach. With the help of a graduate student, the late Clifford Berry, Atanasoff had built a prototype computer by December 1939. A working model of the Atanasoff-Berry computer was completed in 1942. It was a serial, binary, electro-

mechanical machine, and employed various new techniques that Atanasoff had invented, including novel uses of logic circuitry and a regenerative memory.

Atanasoff did not understand completely the extent of the contribution he had made to the advancement of technology and did not anticipate the wide use of computers. IBM and Remington Rand rejected Atanasoff's overtures, and Iowa State -not fully realizing the potential importance of his computer-failed to obtain patent rights. Discouraged by his own attempts to obtain a patent, Atanasoff gave up. He spent the rest of his professional career in various governmental and industrial jobs, including the presidency of two companies he founded, Ordnance Engineering Corporation and Cybernetics, Inc. For his work with the Naval Ordnance Laboratory during World War II, he received the US. Naval Distinguished Service Award in 1945.

Only very recently has Atanasoff achieved recognition as one of the fathers of the digital computer. In a patent infringement suit filed in 1973 by Sperry-Rand against Honeywell, there was testimony by Atanasoff and others concerning the development of the first electronic computer, the ENIAC, by John Mauchly and J. Presper Eckert of the Moore School of Electrical Engineering at the University of Pennsylvania between 1942 and 1946. The decision in this case by Federal District Judge Earl R. Larson concluded that Eckert and Mauchly had derived some of their ideas from Atanasoff, partly as a result of a visit Mauchly made to Atanasoff at Iowa State in 1941.

G. MOLLENHOFF

ATLAS

For article on related subject see **DIGITAL COMPUTERS**: Contemporary and Future.

The Atlas computer was the third in a series of early computers designed in the United Kingdom by a team under T. M. Kilburn in the Department of Electrical Engineering, University of Manchester, in association with Ferranti Ltd. (later ICT Ltd.). Previous systems were the Ferranti Mark I and Ferranti Mark II (Mercury).

Design of Atlas began in 1958, and ultimately three systems, known as Atlas 1, were constructed and installed at the University of Manchester (1962), University of London (1963), and the Atlas Laboratory, Chilton (1963). All were operated until the early 1970s with the Chilton machine being the last to be switched off in March, 1973.

In many respects, Atlas led the way in design of an integrated computer system, combining many novel hardware features with an advanced software operating system. Among the new concepts that Atlas successfully introduced to the computer world were multiprogramming, one-level store, and paging. It was the first major system designed for multiprogramming and was provided with a composite memory, consisting of ferrite cores and magnetic drums linked by program to provide the user with a one-level store. This was achieved by a paging system in which page switching was controlled by a simple learning program, or swapping algorithm. There was also a wire-mesh/ferrite rod (hairbrush) memory of 8,000 words to hold the supervisor. The standard word length was 48 bits, equivalent to one single-address instruction with two modifiers and allowing for up to 2^{20} addresses; 128 index registers were provided. Instructions were normally executed at an average rate of OS ms, about a hundred times faster than the Mercury computer.

The magnetic tape system used 1-in. tapes, although standard OS-in. tapes could also be used. Magnetic disks were not standard, but were fitted later to the Manchester and Chilton machines. Multiple I/O channels provided for both paper-tape and punched-card peripherals as well as line printers.

Other features of the supervisor program, which was produced by a small team under D. J. Howarth (1961-1962, 1962) were the facilities for scheduling and streaming of jobs, automatic control of peripherals, detailed job accounting, and a sophisticated level of operator control. It was normal, with some discretion in selecting the job mix, to obtain 60-80% effective use of the CPU.

A modified version of Atlas, known as Atlas 2, was produced with increased core memory and no magnetic drums (thereby dispensing with paging), the prototype being the Titan computer at the University of Cambridge, which was taken out of service at the end of 1973. Two others in this series were installed: one at the Atomic Weapons Research Establishment, Aldermaston, and one at the Computer-Aided Design Centre, Cambridge. The latter was still in use in 1974.

AUDIO RESPONSE TERMINAL

Although technical and economic reasons, partly due to advances in component manufacture, prevented the Atlas computers from achieving commercial success, they represent an important landmark in the development of advanced computer systems.

REFERENCES

19614962. Howarth, D. J., R. B. Payne, and F. H. Sumner. "The Manchester University Atlas Operating System; Part II, Users' Description," *Computer J.*, Vol. 4. pp. 226-229.
1962. Howarth, D. J., P. D. Jones, and M. T. Wyld. "The Atlas Scheduling System," *Computer J.*, Vol. 5. pp. 238-244.
- 1961-1962. Kilburn, T., D. J. Howarth, R. B. Payne, and F. H. Sumner. The Manchester University Atlas Operating System; Part I, Internal Organisation," *Computer J.*, Vol. 4. pp. 222-225.
1962. Kilburn, T. D., B. G. Edwards, M. J. Lanigan, and F. H. Sumner. "One-Level Storage System," *IRE Trans.*, EC-1 1, Vol. 2. pp. 223-235.

R. A. BUCKINGHAM

AUDIO RESPONSE TERMINAL

For articles on related subjects **see DATA COMMUNICATIONS; INPUT-OUTPUT DEVICES; and TERMINALS.**

Audio response terminals allow data to be entered into a computer through a keyboard similar to a typewriter terminal. However, the output from the computer is in the form of a spoken (audio) reply. This spoken reply is generated by an audio response unit attached to the computer.

Audio Response Units. Basically, there are two techniques that allow a computer to generate a spoken reply. The first technique is to synthesize human speech by the generation of signals and frequencies similar to that produced in speech. Such a device is the IBM 7772 audio response unit (Fig. 1). The other technique is based upon the prerecording on a magnetic drum or disk of words spoken by humans, similar to the use of magnetic tape in home tape recorders. In this case the audio response unit is able to select a number of prerecorded words from the drum or disk, and transmit these one at a

time along a telephone line to an audio response terminal, so constructing a meaningful sentence or message. The IBM 7770 audio response unit uses this technique (Fig. 1). With this unit, the number of words that may be prerecorded is limited by the size of the drum. Such audio response units have a limited vocabulary, but are fully capable of providing meaningful responses when used for specific applications. The appropriate words relating to the particular applications are prerecorded on the drum, generally during manufacture of the unit.

Thus, audio response units allow a computer to respond to a question with a spoken reply in whatever language required and, in the case of the 7770, in any accent, using either a male or female voice as prerecorded on the drum.

Devices suitable for use as audio response terminals include Touch-Tone telephones and portable audio terminals.

THE TOUCH-TONE TELEPHONE AS A TERMINAL. The Touch-Tone telephone (Fig. 2), which uses buttons rather than a dial, is now widely in use. The depression of a particular button creates a tone of an audio frequency that can be transmitted along a telephone line. Each button generates a specific tone, so that each tone can be interpreted as a particular number.

As a computer input device, Touch-Tone telephones are very economical and readily available. However, they are generally limited only to numeric information, plus one or two other special keys or buttons. The general-purpose Touch-Tone telephone does not have the ability to enter alphabetic information. Similarly, the only medium available for output is an audio response in the human speech frequency.

The advent of audio response units provided a means whereby the Touch-Tone telephone could be used both as an input device and an output device. Thus, any Touch-Tone telephone may be used as a computer terminal by first dialing the number of the computer, after which a telephone operator answers the call and connects into the computer the appropriate transmission line used by the telephone. Alternatively, the computer may itself automatically answer the call and connect itself to the appropriate transmission line.

After the call has been established, the numeric keyboard on the Touch-Tone telephone may be used to enter information to the computer—for example, an inquiry requesting the stock availability of specific products. The audio response unit accepts the various tones transmitted along the telephone line and converts these into digital signals representing

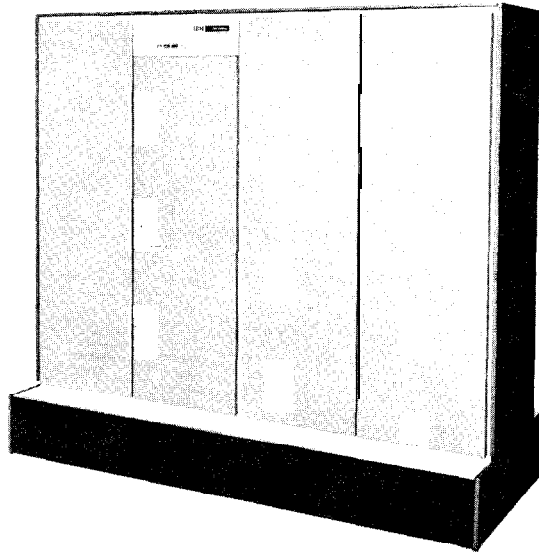
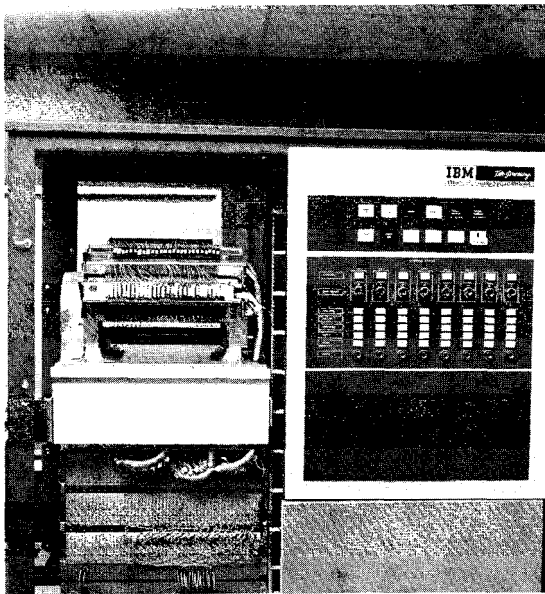


Fig. 1. IBM 7770 (left) and IBM 7772 (right) audio response units.



Fig. 2. Touch-Tone telephone, which may be used as an audio response terminal for numeric input.

the appropriate numeric value. At this stage, the message received from the Touch-Tone telephone is in exactly the same form in the computer as it would

come from a card reader or a typewriter terminal, for example. Consequently, the computer can access the appropriate files, determine the necessary response to the question received from the terminal, and prepare that response for transmission back to the telephone,

However, while the various letters making up the words comprising the reply are transmitted directly to other terminals after translating those letters into the appropriate terminal code, with audio response terminals the process is slightly different. In this case (particularly with the IBM 7770 audio response unit) the response constructed by the computer is a series of addresses rather than words. These addresses indicate the location on drum of the appropriate word to be read and transmitted along the telephone line. The audio response unit recognizes each of these addresses, reads the appropriate words from the specified drum location, and plays back that word on the telephone line, after which it is transmitted to the telephone receiver. As each subsequent word is accessed by the audio response unit, the words combine to form a spoken reply to the request entered by the Touch-Tone terminal.

PORTABLE AUDIO TERMINALS. Portable terminals, designed to increase the capabilities of audio

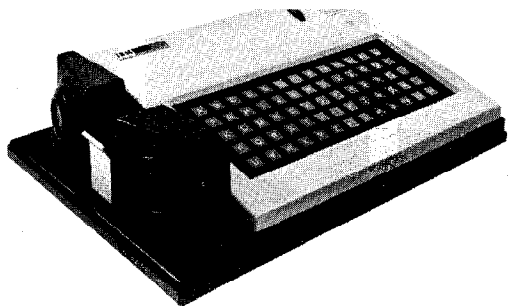


Fig. 3. IBM 2721 portable audio terminal used for alphanumeric input.

response terminals by providing a full typewriter keyboard with alphabetic, numeric, and special character keys, are now available (Fig. 3). These terminals attach to a normal telephone handset, using an acoustic coupler, by placing the handset in a receptacle in the terminal. Thus, signals generated by the keyboard can be transmitted through the mouthpiece of the telephone to the computer, and a spoken response can be received through the telephone receiver and amplified by the terminal if necessary.

Such terminals may weigh as little as 10 lb. (Fig. 3) and can be carried in a small case that contains all necessary batteries and connectors for operation. The terminal can be moved from point to point, and yet is able to enter both alphabetic and numeric information from any telephone, regardless of whether that telephone has Touch-Tone keys or a dial.

Thus, this kind of terminal is particularly suited for applications requiring considerable mobility, such as a representative traveling to various customers. In this case, the representative may take orders for goods from the customer and then use his portable audio terminal to transmit that order directly to the computer. He can use the customer's telephone on his own premises, dial the computer, and then enter the order through the keyboard. The computer is able to access appropriate files to determine the stock availability, for example, of the goods ordered and give a spoken response indicating the acceptance or rejection of the order.

AUDIO TERMINALS USED FOR DATA ENTRY. While audio response terminals have mainly been used in an inquiry environment, they are also suited for use as data entry devices. Because of their low cost and ready availability, such terminals can be used to capture data at its point of origin and

transmit that data directly to the computer. In this way the need for any data transcription is bypassed. Certainly, no hard copy or visual display of the computer response is practical. However, as data is keyed on the terminal and transmitted to the computer, that data can be immediately edited and validated, accessing any computer files that may be necessary to determine the accuracy and reasonableness of the information received. If the data passes the appropriate validation tests, the computer can transmit a spoken response such as "accepted." However, if an error is detected in the data, the computer can transmit a spoken error message, such as "invalid product number-please reenter," or "only quantity of xxxx in stock-enter A to accept or C to cancel."

The Future. While the use of computers for spoken responses to queries is a practical and useful proposition today, at present the use of the human voice for input of data directly into a computer is not a commercial proposition. Although experimental devices have been developed to recognize certain words spoken by a limited number of people, the various intonations, accents, and use of language in the human voice makes audio input a vastly more difficult job than audio output. Nevertheless, the time may come when we can pose questions to a computer in a spoken voice and receive a spoken reply.

C. B. FINKELSTEIN

AUTHORING LANGUAGES AND SYSTEMS

For articles on related subjects see **COMPUTER-ASSISTED INSTRUCTION**; **COMPUTER ASSISTED LEARNING AND TEACHING**; **COMPUTER-MANAGED INSTRUCTION**; and **PROGRAMMING LANGUAGES**.

Considerable attention has been given to providing a convenient programming language for the use of authors of computer-based learning materials. However, obtaining a single, ideal language is a fiction; different uses require different capabilities, which are not conveniently provided within a single language and its associated processor. Furthermore, most users want to learn only those parts of the

language and system which are necessary to accomplish a particular set of purposes.

The specific programming language used by an author is not so significant for effective computer-based instruction as are three other factors. With what notation does the author describe for himself and others the substance and procedures of his computer-based instruction? By what means are these ideas and notes reliably transcribed into an executing computer program? How efficiently does the computer operate on this program and related data bases to deliver instruction or provide adjunct learning activities?

One concept of authoring languages and systems is represented in Fig. 1. The designer of material assembles information and opinion about what is needed, working with students and others who should know of the problems and resources (steps 1 through 3). The designer may work with a language or notation devised especially for the topic and objectives (step 4), delegating to the machine or technical assistants the determination of minor details (step 5). Separation of the content of instruction from the description of program logic makes curriculum development less costly. Various steps in the translation may be handled by humans or by ma-

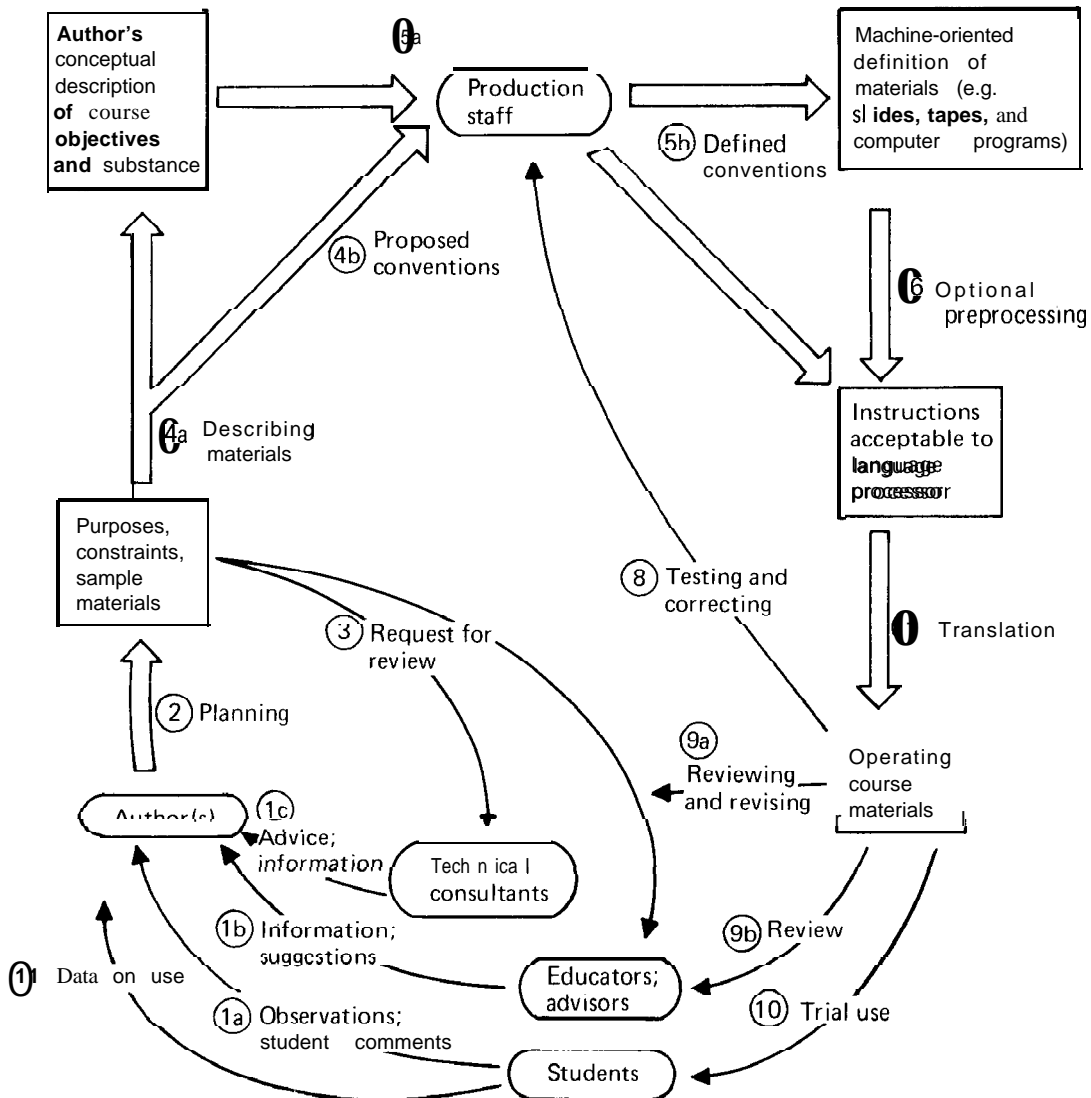


Fig. 1. One representation of authoring activity.

Description of successive frames	COPI (Univac) COURSEWRITER (International Business Machines) IDF (Hewlett-Packard) CAN (Ontario Institute for Studies in Education) SCHOLAR-TEACH (Digital)
Description of interactive case histories	PILOT (University of California, San Francisco) TUTOR (PLATO, University of Illinois) PLANIT (System Development Corporation)
Description of instructional procedures	FOIL (University of Michigan) MENTOR (Bolt, Beranek and Newman)
Specification of data generation and simulated laboratories	TSA (Stanford University) CAL/APL (Coast Community College District) APL/CAT (Erindale College, University of Toronto) Algol, Snobol, Fortran, others
Problem solving and programming (on-line)	EXPER SIM (University of Michigan) APL (International Business Machines) BASIC (Dartmouth Time-Sharing System) LOGO (Bolt, Beranek, and Newman)

Fig. 2. A sample of programming languages for instructional use of computers.

chine (steps 6 and 7), depending on considerations of expense, reliability, and convenience.

After a program is executing, the originator should receive complete and useful information about the performance and reaction of students at various test sites (steps 9 through 11). Many developers continue to test and revise instruction-related computer programs over a long period of time.

Over 50 different languages and dialects have been put to use specifically for programming instructional use of computers. A select set are diagrammed in Fig. 2 to represent different approaches and uses: successive frames (computerization of programmed instruction); description of interactive case histories; description of instructional procedures; specification of data generation and simulated laboratories; problem solving and programming.

<i>Program Statements</i>	<i>Description of Directive</i>
GENERAL: "Proceed with investigation."	[type out message within quotation marks for the student to read: "GENERAL" labels this block of the program]
ACCEPT	[accept directive from student and execute statements which are indented below]
IF /suspects/ (REPORT I)	[if the student mentioned "suspects" execute (I) or (2) indented below]
1) "Wife, brother and partner."	[first time, type message marked (I)]
2) "No new suspects."	[all other times, type message marked (2)]
IF /lab, rifle, glass, pipe/	[if he mentioned any of these, execute the following]
IF ALL REPORTS, GO TO LAB	[only if he has requested all reports, defined elsewhere, process his request for a laboratory test]
"I advise you to check reports first."	[if he has not requested all reports, advise him accordingly]
IF /interrogate/	
IF ALL LAB, GO TO INTERR	[if all lab tests have been requested, process his request for interrogation]
"I advise you request lab tests first."	[otherwise, advise . . .]
• GLASS	
• • •	
"I don't understand."	[if nothing is recognized in his directive let him know and wait for another directive]
LAB "This is the lab."	
IF /glass/ (GLASS)	[if he mentioned the "glass" set GLASS switch]
IF WIFE	[and if he has already interrogated the wife]
"Glass contained arsenic."	[type damaging evidence]
1) "Prints belong to the wife."	[otherwise type (1) the first time]
2) "Nothing new."	[type (2) all other times]
• • •	
• • •	
"What is it you want?"	[if no lab test request is recognized, type query]
ACCEPT	[accept directive]
TO LAB + I	[go back to process a second directive relating to the lab]

Fig. 3. Sample of a notation suited for exercises in information gathering and diagnosis (interactive case histories). A sample conversation of this program is given on p. 269.

AUTOMATA

Information about these and many other languages can be obtained from secondary sources (Zinn, 1971; Bode and Dutting, 1974).

A sample of program code for an interactive "case history" is given in Fig. 3. The annotations in the right column point out the means by which the author specifies contingencies conveniently in the interaction between user and program. The example is adapted from a Mystery problem programmed in the Mentor language; both were developed at Bolt, Beranek, and Newman.

Any language appears simple and convenient when it is used for the specific purpose for which it was designed to be convenient. The great diversity of instructional uses of computers repeatedly forces specific-purpose languages into situations for which they were not designed.

REFERENCES

1971. Zinn, Karl L. "Requirements for Programming Languages in Computer-Based Instructional Systems." Paris: Organization for Economic Cooperation and Development.
1974. Bode, Arndt, and Martin Dutting. "Computer-Assisted Instruction: Problems, Languages, Systems, and Documentation" (translated from the German and edited by Karl L. Zinn). Ann Arbor: EXTEND Publications.

K. L. ZINN

AUTOMATA. See **CELLULAR AUTOMATA**; and **PROBABILISTIC AUTOMATA**.

AUTOMATA THEORY

For articles on related subjects see **ALGORITHM**; **CELLULAR AUTOMATA**; **FORMAL LANGUAGES**; **PERCEPTRON**; **PROBABILISTIC AUTOMATA**; **SEQUENTIAL MACHINES**; and **TURING MACHINE**.

For articles on related terms see **CONCATENATION**; and **PROGRAM**.

Introduction and Definitions. Automata theory is a mathematical discipline concerned with the invention and study of mathematically abstract,

idealized machines called "automata." These automata are usually abstractions of information processing devices, such as computers, rather than of devices that move about, such as mechanical toys or automobiles.

This article gives a short and informal survey of the major classes of automata that automata theorists have heretofore seen fit to study, and indicates the primary respective motivations (from the point of view of computer science) for the study of these classes of automata.

For the most part, the automata discussed here process strings of symbols from some finite alphabet of symbols. Let A be any alphabet (finite set of symbols). For example, A might be $\{a, b, c, \dots, z\}$ or $\{0, 1\}$. We write A^* to mean the set of all finite strings of symbols chosen from A . If A is $\{a, b, c, \dots, z\}$, then A^* contains strings representing English words, such as "cat" and "mouse," along with nonsense strings such as "czzzxyh". If A is $\{0, 1\}$, then A^* contains the strings representing the non-negative integers in binary notation ($0, 1, 10, 11, 100, \dots$) and at, these same strings but with extra zeros on the left (e.g., 00010).

Automata generally perform one (or both) of two symbol-processing tasks. They compute partial functions from X^* to Y^* for some finite alphabets X and Y or they *recognize* languages over some alphabet X .

A partial function from X^* to Y^* is a correspondence between some subset of X^* and the set Y^* that associates with each element of the subset of X^* a unique element in Y^* . For example, let $x = Y = \{0, 1\}$ and let the subset of X^* be the elements x of X^* such that x begins with 1 or consists of a single 0. If f associates with x the string in Y^* that denotes the binary number representing two times the binary number represented by x , then f is a partial function from X^* to Y^* .

We say roughly that an automaton α *computes* a partial function f from X^* to Y^* when, if α is given any input x in X^* such that $f(x)$ is defined, α eventually produces an output $y \in Y^*$ such that $f(x) = y$, and, otherwise, α produces no output. Automata usually receive their inputs on a linear or one-dimensional tape, which they are capable of reading one symbol at a time. The manner in which they read symbols on an input tape (left to right, back and forth, with or without changing symbols, etc.) depends on the particular class of automata under consideration. Automata for computing partial functions produce their output on a tape (perhaps the input tape, perhaps a different tape) in a manner also prescribed by the particular class of

automata under consideration.

A language over an alphabet X is just a subset of X^* . For example, if $X = \{a, b, c, \dots, t\}$, then $\{a, aa, aaa, \dots\}$ and $\{x \in X^* \mid x \text{ is a word in the English language}\}$ are both languages over X .

We say that an automaton α recognizes a language L over X when α reads an input $x \in X^*$ on its input tape in the manner of automata of its type; then, if $x \in L$, α eventually performs some particular act of recognition such as halting, emptying a particular auxiliary tape, or getting into some special internal state; whereas, if $x \notin L$, α never performs such an act of recognition. Exactly what constitutes an act of recognition depends on the particular class of automata under consideration.

It is presumably clear why it is of interest to computer scientists to study automata that compute (partial) functions, since computer science is the computation business. Among the interesting questions to ask are whether some function is or is not computable by some representative of a particular class of automata and, if it is computable, how efficiently (with respect to some mathematically precise measure of efficiency) can it be so computed.

We motivate the study of automata that recognize languages by some examples. Let X be the set of allowable symbols for some programming language P . Include in X the necessary punctuation symbols and the blank symbol. Let $L = \{x \in X^* \mid x \text{ is a valid program of } P\}$. In the process of compiling from P into some other language, it is useful to (among other things) recognize the valid programs of P as being valid. Automata theory gives some insight into the sort of computing ability that may be required to recognize valid programs. For example, pushdown **automata** (to be defined below) are capable of recognizing the valid syntactic classes of all (and **only**) Algal-like languages.

Automatic theorem proving, a subarea of artificial intelligence, is also concerned with language recognition. The language to be recognized is the set of propositions derivable from some set of axioms. Automatic theorem proving has been applied to discovering new mathematical theorems, to question-answering systems, and to robotics.

Types of Automata. Most (but not all) types of automata are special cases of the Turing machine (see Fig. 1). Turing machines may be operated either to recognize or to compute partial functions. Very

roughly, a Turing machine is a finite-state deterministic device with read and/or write heads (which read and/or write one symbol at a time) attached to one or more tapes. "Finite state" means that the number of distinguishable internal configurations of the device is finite, and "deterministic" means that the next state of the device and its subsequent action (writing or motion) on the tapes is completely determined by its current state and the symbols it is currently reading on its tapes.

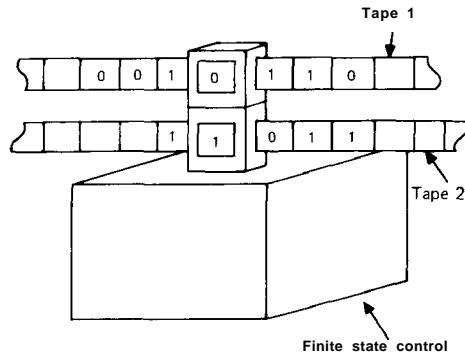


Fig. 1. A two-tape Turing machine. Each tape is scanned by a single read/write head. Tape 1 contains the string of nonblank symbols 0010110, with the underlined 0 currently being read. Tape 2 contains 11011, with the underlined 1 being currently read. If the tapes can move only in one direction, the same diagram would depict a two-tape automaton.

Turing machines were first introduced independently by Turing and Post in 1936 to give a precise mathematical definition of **effective procedure**. There is considerable evidence that the partial functions computed by (languages recognized by) Turing machines are exactly those computed (recognized) by informal effective procedures or algorithms. Any computation or recognition problem for which there is a known informal algorithm can be handled by a Turing machine. Turing machines with many (in general, n -dimensional) tapes and read/write heads can compute and recognize no **more** than can Turing machines with a single one-dimensional tape and single read/write head, although they may compute and recognize more efficiently.

Attempts to define **effective** procedures in terms of automata more closely resembling modern electronic stored-program digital computers have led to the unlimited register machines of Shepherdson and

AUTOMATA THEORY

Sturgis and to the *random-access* stored-program machines of Elgot and Robinson. These machines can be shown to compute the same partial functions (recognize the same languages) computed by Turing machines.

Turing machines model the most general sort of computation processes, in part by virtue of their ability to move about freely on their tapes without fear of running out of tape. In general no a priori bound can be set on the amount of tape a Turing machine computation will require. Some Turing machine computations may require more tape than is available in the universe! This, in part, motivates our consideration of the next class of automata, finite automata. We will limit our discussion to finite automata considered as recognizers of languages, and will leave their application as input/output devices to the article on sequential machines.

A finite automaton is a deterministic finite-state device equipped with a read (only) head attached to a single input tape. A special subset of the finite set of states of a finite automaton is designated as the set of *final*, or *recognition*, states. A finite automaton α processes a string of symbols thus: α begins in a special initial, or start, state and automatically reads the symbols of x (on its tape) from left to right, changing its states in a manner depending only on its previous state and the symbol just read. If, after the last (rightmost) symbol of x is read, α goes into a final state, α recognizes x ; otherwise, α does not recognize x . Let $A = \{0,1\}$. It is possible, for example, to design a finite automaton α such that α recognizes $L = \{x \in A^* \mid x \text{ ends in two consecutive 1s and does not contain two consecutive 0s}\}$. See Fig. 2. On the other hand, it can be shown that *no* finite automaton can recognize $L' = \{x \in A^* \mid x \text{ consists of a consecutive string of } n\text{-squared 1s for some positive integer } n\}$. As might be expected, however, a Turing machine can be designed to recognize L' .

In Fig. 2 the circles represent the different states of α_0 and the number inside each circle is a name for the state that circle represents. Hence, 0 is the start state of α_0 and 4 is its only final state. An arrow (labeled with an alphabet symbol) from one state to another means that if α_0 is in the first state while scanning the alphabet symbol that labels the arrow, then it goes next into the second state. For example, if α_0 is in state 1 scanning a 0, it goes next into state 2; whereas, if it is in state 1 scanning a 1, it goes next into state 3. If α_0 is given the input string 010111, beginning in state 0, the successive states into which it is thereafter driven are (in order) 1,3,1,3,4,4. Since 4 is a final state, α_0

(correctly) recognizes the input string 010111. If α_0 is given 10011, beginning in state 0, the successive states into which it is thereafter driven are (in order) 3,1,2,2,2. Since 2 is not a final state, α_0 (correctly) fails to recognize 100 11.

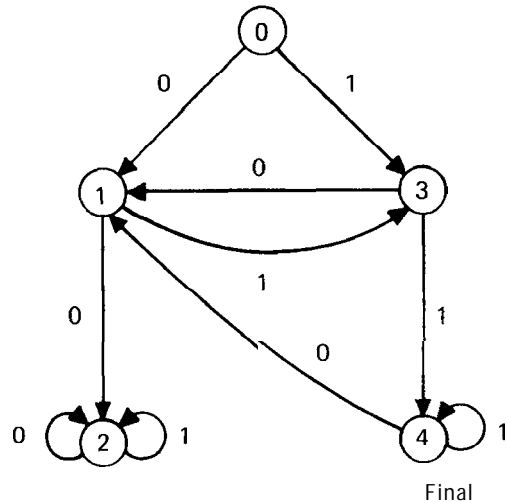


Fig. 2. The state diagram of a finite-state automaton for recognizing $\{x \in \{0,1\}^* \mid x \text{ ends in two consecutive 1s and does not contain two consecutive 0s}\}$.

A nondeterministic finite automaton is a device just like a finite automaton except that the next state is not completely determined by the current state and symbol read. Instead, a set of next *possible* states is so determined. A nondeterministic finite automaton α may be thought of as processing a string of symbols x , just like an ordinary finite automaton except that it has to be run over again several times so that each of the different possible state-change behaviors is eventually realized. One should imagine there being a separate, deterministic control device C which runs α and completely determines α 's *actual* state-change behavior each time it is run. There are but finitely many different possible state-change behaviors for α processing x , and C simply systematically runs α first one way, then another, then another, etc., until all possibilities have been exhausted.

A finite automaton α recognizes x just in case at least one of the possible ways of running α on input x results in getting α into a final state after the last symbol of x has been read (see Fig. 3).

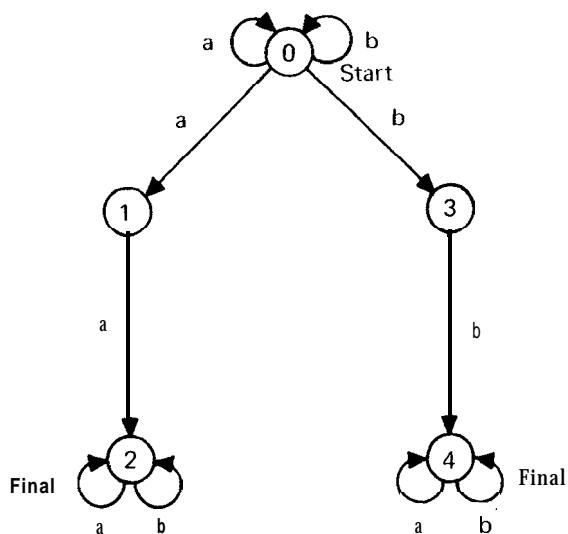


Fig. 3. The state diagram of a nondeterministic finite-state automaton α_1 for recognizing $\{x \in \{a,b\}^* \mid x \text{ contains two consecutive } a\text{'s or two consecutive } b\text{'s (or both)}\}$.

In Fig. 3, α_1 is nondeterministic because (for example, from state 0, if it is scanning a) it can go into either state 0 or state 1 next. From state 1, if it is scanning b , it "jams," since the set of next possible states is empty. If α_1 is given the input string $abababba$, beginning in state 0, one possible succession of states is (in order) 0,0,0,0,0,0,1. Here, 1 is not one of the final states, so this way of running α_1 does not lead to recognition. Another possible succession of states is (in order) 0,0,0,1, jam. Another is 1, jam. However, 0,0,0,0,0,3,4,4 is still another possible succession of states. Since 4 is a final state, α_1 (correctly) recognizes $abababba$. It is easy to check that if α_1 is given $babababab$, beginning in state 0, then none of the possible ways of running α_1 leads to a final state; hence, α_1 (correctly) does not recognize $babababab$.

Interestingly (and perhaps unexpectedly), it can be shown that nondeterministic finite automata recognize exactly the same class of languages as ordinary finite automata. Turing machine recognizers that operate nondeterministically can also be defined, but they cannot recognize more languages than can ordinary Turing machines. For nondeterministic Turing machine recognizers, as well as for some of the other nondeterministic devices to be discussed below, some of the different possible ways to process a given string x may take infinitely many

steps. For such devices it is convenient to imagine the separate, deterministic control device C as operating in a parallel mode.

In addition to ordinary and nondeterministic automata, a variety of automata called "probabilistic" automata have been studied. A probability of occurrence is assigned to each of the possible next states in a probabilistic automaton.

In 1943 McCulloch and Pitts introduced nets of formalized neurons and showed (in essence) that such neural nets could realize the state-change behavior of any finite automaton. These nets were composed of synchronized elements, each capable of realizing some boolean function such as AND, OR, or NOT. It has been suggested that von Neumann had these networks in mind when he established his logical design for digital computers. In 1948, von Neumann added to the computational and logical questions of automata theory by introducing new questions pertaining to construction and self-replication of automata. The iterated arrays of interconnected finite automata which he introduced have also been used to study pattern processing for patterns of symbols, including (but not restricted to) one-dimensional strings of symbols.

Automata theory, especially finite automata theory, impinges on both mathematical systems theory and modern algebra. In mathematical systems theory, one is interested in the problem of which, (if any) input sequences will drive an automaton to some desired internal state. In modern algebra one can study the relations between semigroups and automata. For example, certain decomposition theorems in group theory give information about decomposition of automata into particularly simple component automata.

A linear-bounded automaton is a nondeterministic, one-tape Turing machine whose read/write head is restricted to move only on the section of tape initially containing the input. Special end markers are placed on each side of an input string to prevent the tape head from leaving this restricted section of tape. A form of deterministic linear-bounded automata was first studied by Myhill in an attempt to find models of computation more realistic than the completely general Turing machines, but less restricted than the finite automata. Later it was shown that linear-bounded automata recognize all (and only) the *context-sensitive* languages, an important and natural class of languages more restricted than the languages recognizable by Turing machines but

AUTOMATION

more general than the *context-free* languages. It is an open question whether the linear-bounded automata can recognize more languages than the deterministic linear-bounded automata.

A *pushdown* automaton is a nondeterministic finite automaton with a special sort of auxiliary tape called a "pushdown store." A pushdown store is a tape quite like the stack of plates found on a spring in cafeterias. It is a "Last In-First Out" store. A special read/write head always scans the top symbol on the pushdown store. The pushdown store is initially loaded with a single special **START** symbol. The top symbol can be replaced by any finite string of symbols (stack of plates), including the empty string of symbols. Replacing the top symbol by the empty string has the effect of completely removing the top symbol and setting the read/write head to scan the next symbol down. The read (only) head on the input tape reads one symbol at a time from left to right, just as in a finite automaton, except that it is allowed (if desired) to stop scanning the input tape momentarily while only the pushdown store is operated.

Pushdown automata recognize a string x by one of two conventions. Either x is recognized by the device as it gets into one of its final states or by the pushdown store as it empties just after the rightmost symbol of x is read. The class of languages recognized by emptying the pushdown store is the same as that recognized by final states. Let $A = \{0,1\}$. For $x \in A^*$, let x^R be x written backwards. For example, 001110^R is 011100 . $L = \{x \in A^* \mid x \text{ is of the form } w \text{ followed by } w^R \text{ for some } w \in A^*\}$ is recognizable by a suitable pushdown automaton; however, L is *not* recognizable by any finite automaton or even by any deterministic pushdown automaton. Pushdown automata recognize all (and only) the context-free (or equivalently, Algol-like) languages.

Many variations on a slight generalization of pushdown automata have been studied. A *stack* automaton is just like a pushdown automaton except that the read (only) head of the input tape is allowed to move both ways (but not off the section of tape containing the input) and the read/write head on the pushdown store is allowed to scan the entire pushdown list in a **READ ONLY** mode. The class of languages recognized by stack automata is intermediate between context-sensitive and Turing-machine recognizable.

Many other types of automata that have been and could be studied employ some other sort of

limited data structure for their auxiliary storage or receive inputs in some form other than a string of symbols. For example, *tree* automata process inputs in the form of trees, usually trees associated with parsing expressions in context-free languages.

It should be remarked at the conclusion of this survey that automata theory is a growing, open-ended mathematical discipline. It readily admits of extensions of existing concepts and the introduction of totally new ideas. The motivations to make such extensions are esthetic on the one hand, and the need or desire to model some existing or proposed computational phenomenon on the other.

REFERENCES

- 1966. Von Neumann, J. *Theory of Self-Reproducing Automata* (edited and completed by A. W. Burks). Urbana: University of Illinois Press.
- 1967. Minsky, M. *Computation: Finite and Infinite Machines*. Englewood Cliffs, N.J.: Prentice-Hall.
- 1969. Arbib, M. A. *Theories of Abstract Automata*. Englewood Cliffs, N.J.: Prentice-Hall.
- 1969. Hopcroft, J. E., and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.
- 1969. Minsky, M., and S. Papert. *Perceptrons*. Cambridge, Mass.: M.I.T. Press.
- 1973. Bobrow, L. S., and M. A. Arbib. *Discrete Mathematics: Applied Algebra for Computer and Information Science*. Philadelphia: W. B. Saunders.
- 1973. Engeler, E. *Introduction to the Theory of Computation*. New York: Academic Press.

J. CASE

AUTOMATION

For articles on related subjects see **COMMUNICATIONS AND COMPUTERS**; **COMPUTER NETWORKS**; **COMPUTERS AND SOCIETY**; **CONTROL APPLICATIONS**; **DIGITAL COMPUTERS**; and **TIME SHARING**.

For articles on related terms see **INFORMATION SCIENCE**; **MINICOMPUTERS**; **OPERATIONS RESEARCH**; **OPTICAL CHARACTER**

READERS; SIMULATION; TERMINALS; and
COMMUNICATION CONTROL UNIT.

Concept of Automation. The concept of automation includes five main functions

1. Collection of information through data collection equipment.
2. Communication between man-machine, man-man, and machine-machine, through generating and regulating the flow of data collected.
3. Computation with the information, such as data logging, data analysis, and data processing with the help of mathematical formulations.
4. Control of operations, both human and mechanical, on the basis of information analysis.
5. Logical coordination among the preceding four functions.

In order to describe the different functions of automation, the following key words might be used:

- Information *collection* function: based on measuring devices of physical (time, length, weight, temperature, etc.) and chemical properties (concentration, color, etc.), counting devices, reading devices (optical character reading) and data preparation.
- Information *communication* function: based on data communication, computer networks, using different types of languages for communicating data within man-machine systems (problem-oriented languages).
- *Computation* function: based on information processing, data processing, data management.
- Operation *control* function: based on process control, process simulation, cybernetic features (feedback and feed-forward control), distributed control.
- *Coordination* function: based on management information systems, computer-aided instruction, operations research, and model building.

The automation concept is thus not limited to a given production process or a given service but is linked to a host of human activities. Attempts have been made to define certain degrees of automation; as a rule these apply to the labor intensive aspect of a given manufacturing industry. However, no general method for assessing automation in its broad sense has yet been developed.

In the absence of a general statistical framework for assessing the diffusion of automation in the

different sectors of the economy, the number of computers used in a given sector will indicate, to a certain extent, the progress made in automating its activities. There are, of course, many industrial processes or services that can be considered highly automatic without having to rely on computing facilities, especially when information processing is not critical to fulfillment of the process or service. This is especially true for specialized small-scale industries and service centers (e.g., precision instrument assembling and automatic vending machine centers) where physical communication, information flow, and data processing are of minor importance in the total activities. However, for those production processes and services that require a large, complex organization, and a relatively large amount of information processing (and the general trend is a continuing shift in this direction as a consequence of large-scale, multinational manufacturing and services), the use of computers becomes increasingly critical with regard to automating these activities. In other words, the more an activity increases in volume and importance, the more the criterion of computer utilization, as an indicator of the degree of automation, becomes relevant.

According to available statistics on computer utilization, the largest share of computers is used in the metal-working industry, including both mechanical and electrical engineering (20%); banking, insurance, administration, research and development, and computing centers each account for 10% of the computer market. The chemical and food industries and trade and transport each account for 5% of the market, and the remaining 10% is distributed between other industrial sectors and services. See Table 1.

The main functions of the computer are financial applications (30%) production control (20%), market studies (15%), distribution (15%), research and development (10%), and coordination (10%). See Table 2.

The aim of automation, in its broad sense, is thus to enhance human capabilities, both manual and intellectual, in order to meet the increasing complexity of the human way of life. In the future, emphasis will be laid on a global approach to solve problems facing the manufacturing and servicing activities that result from the interaction of technological, economical, and social forces. Such a global approach is only possible if the relevant information is adequately collected, rapidly and correctly diffused, and logically processed in order to serve as guidelines for controlling and coordinating forces.

AUTOMATION

Table 1. Distribution of Computers in the Different Economic Sectors (U.S., %).

Industry	40
Banking and insurance	20
Administration	7
Transport	7
Processing services	6
Research	5
Public services	5
Miscellaneous	10

Table 2. Main Functions of Computer Utilization (U.S., approx. %).

Financial applications (payroll, etc.)	30
Production control	20
Market surveys	15
Distribution	15
Research and development	10
Coordination, scheduling, forecasting	10

Current State of Automation

TECHNOLOGICAL DEVELOPMENT. The progress of automation is a result of scientific research and technical development in the following main sectors:

- Electronics industries (primarily for information collection).
- Telecommunication equipment industries (primarily for information flow).
- Computing equipment industries (primarily for information processing).
- Industrial control industries (primarily for process control).
- Management services (primarily for coordinating activities).

The most striking technical achievements in these sectors in the past years are the following:

1. In the electronics industries, almost any type of physical or chemical value can be measured by highly reliable sensors, based essentially on the progress made in solid state electronics, resulting in miniaturization and high reliability of semiconductors and integrated circuits.

2. In the telecommunication equipment industries, outstanding achievements have been made in large-scale systems such as the worldwide communication network through satellites and the inter-continental computer networks. The development of

transmission control units, especially as regards data concentrators and message switchers in conjunction with the development of numerous computer terminals, has greatly enhanced the possibilities for information flow in time and in space.

3. In the computing equipment industries great developments have been made both in hardware and in software. Minicomputers and supercomputers have increased the equipment range, the former being more or less tied into the communication equipment and linked to data preparation and data reduction devices and used to control various peripheral equipment. The development of time sharing has led to the proliferation of computer terminals and has made possible the efficient use of **supercomputers**. Optical character reading and the use of packaged programs have facilitated the utilization of computers in a great variety of nonmanufacturing activities such as wholesale and resale trade, and personal services.

4. In the industrial control industries, which include electrical measuring instruments, engineering and scientific instruments, measuring and controlling instruments, automatic temperature controls and optical instruments, the reliability and miniaturization of the equipment, by using integrated circuits as well as the direct link of the equipment onto a computer (direct digital computer control), are the **most outstanding** development features.

5. In the field of management services, research in information science, management science, operations research, and software management has permitted greater insight into computer aided manufacturing and servicing, thus increasing the efficiency of computer utilization and automatic control. But, above all, the spread of automation has resulted in a systematic collection of the basic information of a large number of industries and services that will be used not only for running the production process (or service) itself but also for analyzing the prospective development of the processes or services, through simulation techniques and long-range planning analysis.

So far, there are about 500 packaged programs on the market which are readily available to users for commercial, scientific, and management applications. Their number will rapidly increase in the near future.

In the coming years the pace of technological progress and technological change will be significantly increased through the greater acquisition, dissemination, and application of scientific and technical knowledge, mainly thanks to automation



Fig. 1. IBM System/7, an automated laboratory aid.

and computer techniques. It is generally assumed that the rate of development and diffusion of technological innovations introduced during the post-World War II period (1945-1960) was twice the rate for those introduced during the post-World War I period (1920-1940) and three times the rate for innovations introduced during the early part of the century (1890-1920).

Not only the pace of technological progress and change but also its diffusion from advanced technological sectors into less developed industries will increase in the future. This accelerated generation and transfer of technology will speed the rate of economic growth. As a consequence, management will become more future-oriented and planning-minded in order to remain at the vanguard of development. This requires adequate information processing, in which the computer will play a dominant role in the future. The efficient use of computers in education and management might thus be considered as the dominant criterion for coping with the pace of technological progress.

ECONOMIC CHANGE. Four main characteristics of computer applications explain their rapid growth

rate in all industries and services:

1. They are time-saving, thus influencing productivity directly.
2. They have a very large capacity for data handling, thus making possible the accurate control of large-scale systems.
3. They can work on many different applications at the same time, thus permitting a perfect integration of data processing, handling, and distribution of information without delay.
4. The computer programs developed for general-purpose applications, and which are readily available in program libraries, can be adapted to specific requirements. Therefore they permit a rapid introduction of computer technology into a large variety of industries and services.

The development of many traditional industries and the creation of many new industries and services were thus only made possible through the introduction of computer-based automation. Large nuclear powerplants, for instance, as well as the present air transport system, would be impossible to operate

AUTOMATION

Table 3. Forecast of the Economic Development of Industries Related to Automation (U.S.)

(1) Industry and Sector	1970		1975		1980
	(2) Value of Shipments, billion \$	(3) Number of Establishments	(4) Number of Employees (1,000)	(5) Expected Value of Shipments, billion \$	(6) Expected Value of Shipments, billion \$
Electric equipment and components	22.5				
Consumer products	3.0	350	140	30	45
Telephone and telegraph	4.0	90	170	4.5	6.0
Electronic equipment	8.5	640	330	6.5	10.5
Electronic components	5.5	1400	340	11.5	17.5
Research and development	1.0			6.5	7.5
Instrumentation (measurement, analysis, control)	5.5			1.0	1.0
Electric measuring devices	1.5	540	70	7.5	10.0
Engineering and scientific instruments	1.2	680	70	2.0	3.0
Measuring and controlling equipment	1.5	660	70	2.0	2.5
Automatic temperature control	0.5	100	40	1.0	1.5
Optical instruments	0.5	300	20	1.0	1.0
Office and computing machinery	5.0	600	190	9.5	13
Typewriters	0.5		—	0.6	0.7
Electronic computing equipment	3.8	—	—	7.5	10.5
Calculating and accounting machinery	0.5	—		1.0	1.5
Office machinery	0.5	—	—	0.5	0.5
Communication equipment					
Domestic telephones	20*) 2000		30'	50*
Domestic telegraph	0.5*)	970	0.5'	1.0*
Radio broadcasting	1.0*	4**	75	1.5*	2.0*
TV broadcasting	3.0'	4**	60	4.5''	6.5''

* Revenues.

** Nationwide networks.

SOURCE: "U.S. Industrial Outlook 1972 with Projections to 1980," U.S. Department of Commerce, Washington, D.C.

without the assistance of computer control. It can thus be said that a substantial part of the development of the American economy, both in volume of production as well as in quality of production, is virtually due to computer applications. It is expected in the near future that in all modern enterprises, the collecting, processing, and distributing of information through computers will increase in volume and in quality, with a direct beneficial effect on production costs and productivity.

The economic impact of automation is felt both in the field of capital spending for equipment and in the pattern of production and product range. The purchase of capital goods related to automation increased significantly in recent years and still shows a significant growth rate in the United States. See Table 3. Many new industries have been created in

order to produce a wide selection of compatible automatic equipment that can be used in a great variety of combinations to meet any specific need.

For example, the computing equipment industry has developed about 250 central computing systems, about 150 varieties each of memory storage and magnetic tape systems, and about 100 different card punch and printer systems. Thus, a great combination of computing equipment is possible to allow optimum selection to meet a variety of needs.

By 1980 it is expected that about \$100 billion worth of electronic equipment will be used in the world, of which almost 60% will be used in industry, over 20% in research and development, and the remaining 20% or less for consumer durables.

The production pattern is, of course, steadily adapting itself to the potentialities of automation

equipment, especially with regard to the optimum number and size of equipment and its required output, to enable automatic equipment to be applied efficiently. Automation generally widens the possible range of plant and equipment size, mainly toward larger units, but sometimes also toward smaller units, especially when it permits the replacement of batch-type processes by continuous-flow processes.

Product range is also significantly influenced by the introduction of automation equipment, both in quality and quantity of the products. Product quality is closely checked for its reliability and its service life, and new products are being marketed at a much faster rate than before. Production costs are more predictable and less influenced by physical production factors, since labor costs influence production rate and production flow to a lesser degree.

The service sector is especially becoming increasingly engaged in automation applications. Many data management institutes offer computer-aided bookkeeping services as well as access to specific data banks for information retrieval. These "computing utilities" already have a \$2 billion turnover and present a growth rate of about 25% per year. In the United States a large computer manufacturer runs a computing center, grouping over 100 large-scale computers with almost 100 regional offices throughout the country. Another manufacturer has established a time-sharing system to which over 40,000 customers have access. About 400 organizations are thus engaged in this market, and an Association of the Data Processing Service Organization (ADAPSO) has been created.

The best example of the possibilities offered today in telecommunication services is the world network of communication satellites. In transport services, computer-controlled flight reservation systems, as practiced by all the major airline companies, are characteristic of the potentialities in traffic organization and control. Finally, with respect to retail sales, it may suffice to indicate the tremendous growth of automatic vending machines for food, drink, stamps, tickets, etc.

Although the rate of development and diffusion is increasing, the elapsed time from the basic technical discovery of technological innovations to the point where economic changes become evident is relatively long. Almost without exception, those technological innovations that will have a significant impact on the economy and on society during the next five years have already been introduced as commercial products or processes. This is especially so for automated equipment, which will not change radically in essence, but rather in a qualitative

manner such as decrease in specific cost and in maintenance, increase in numbers, capacity, speed, and reliability. The economic development of those branches of the industry related to automation will certainly be bright (Table 3).

Also, new industries will emerge as a result of technological change, and diversified services will become available in the economy. Automation will check the self-induced information explosion and paperwork through progress made in data reduction and in computer output on microfilm (COM). A new industry of knowledge will emerge and *information* cost will have to be considered just as carefully as traditional capital, material, and labor costs in the total cost of production. The profit margin will have to be calculated, not only to give a return on the invested capital and to remunerate labor, but also to forecast the technological risk, which will be increasingly important in the industries of the future. The automation-oriented industries will soon reach the level of such basic industries as automobile, petroleum, chemical, and steel. (See Table 4.)

Besides the development of automation in industrial production and in the service sector, a large field of activity is also gradually opening in the management activities. This will be brought about by the introduction of integrated information, computation, and communication systems that will be necessary to apply automation advantageously. The advantages of automation in this sector result from the increased rapidity in the flow of information and in decision making, compact information, and regularity of information, which are necessary in the rapidly changing industrial world. The convergence of political systems will result in a greater interdependence of nations and in a rapid development of multinational corporations and organizations. A prerequisite for this development will be the establishment of international information transfer and processing through automated equipment. For example, General Electric has recently put into operation an international computer system network linking over 300 cities worldwide.

Adequate management techniques and accurate production forecasting will reduce more and more the amplitudes of the economic business cycles and will thus contribute significantly to a balanced economic growth.

THE SOCIAL IMPACT. With a rapidly changing economy made possible by automation, significant changes in many fields are taking place in the work environment and all these within the period of an individual's lifetime. The increase of labor productivity through automation and technological change

Table 4. Comparison of U.S. Industrial Corporations (1971).

Activity	Company	Total assets (A), in billion \$	Operating revenues (R), sales, in billion \$	Manpower (M), in thousand hr	$R_1 = A/M$ in thousand \$	$R_2 = R/M$ in thousand \$
1. Automobile	General Motors	18.0	28.0	733	23	36
	Ford	10.5	16.0	433	24	37
2. Petroleum	Standard Oil	20.0	18.7	143	140	130
	Mobil Oil	8.5	8.2	75	110	110
3. Steel industry	U.S. Steel	6.4	4.9	183	35	27
	Bethlehem Steel	3.4	2.9	115	30	25
4. Chemical industry	Dupont	3.9	3.8	106	37	36
	Union Carbide	3.5	3.0	99	35	30
5. Electric al engineering industry	General Electric	6.8	9.4	363	19	26
	Westinghouse	3.5	4.6	180	19	25
6. Electronic engineering industry	ITT	7.6	7.3	398	19	18
	RCA	3.0	3.7	118	25	31
7. Computer (and related) industry	IBM	9.5	8.2	265	36	31
	Honeywell	2.1	1.9	94	22	20
8. Retail services	Sears Roebuck	8.3	10.0	365	23	27
	Penney	1.9	4.8	162	12	30
9. Telecommunication services	American Telephone and Telegraph	54.0	18.5	776	70	24
	General Telephone and Electronics	9.0	3.8	173	52	22
	Penn Central	4.6	1.7	82	56	21
10. Transportation services	TWA	1.4	1.2	58	24	21

Notes: Based on two largest companies in each field of activity. R_1 = assets per employee. R_2 = revenues per employee.

SOURCE: *Fortune*, June 1972.

results in a shift in the demand for manpower, both in number and in work qualifications. Increased productivity has more impact on employment in industry than in areas of increasing demand for services. Clerical rather than service functions of office workers will be more affected by technological change. The need for professional, scientific, and technical personnel grows at a faster rate than the total labor force, and these personnel require continuing training or retraining to keep their qualifications up to date.

Unemployment will not necessarily result from automation, but there is a real threat of unemployment in those industries that have not kept abreast of technical development and which are bound to go out of business unless they adopt to more efficient operation. Technological unemployment is a temporary situation for a certain number of individuals who did not make a timely change from a traditional activity to a modern one. A person who is thus temporarily out of a job can be compared to the traveler who changes trains to take a new direction but who unfortunately misses the connection. A later

train will surely arrive, but until it does, the traveler has to sit in the waiting room. Compare this situation with that of a traveler on a train shunted onto a dead-end track because it missed the technological crossing and had no other place to go.

Many solutions have been put forward to reduce technological unemployment in the short term; their intent is to reduce the number of people who have missed the train, as well as to reduce the waiting time. Basically, efforts have to be made in two directions: better information and better retraining of the work force while on a job; and better information and better planning with respect to alternative activities.

Here, again, automation will provide part of the answer to improve the situation through better information processing and information transmission, on the one hand, and on the other hand, through improving the training or retraining facilities (audiovisual means of learning, computer-aided training, etc.).

While at the beginning of the century the main shift in manpower was from agriculture to manu-

Table 5. Change in Share of Total Employment by Sector, 1950–1970.

Country	Agriculture	Industry		
		Total	Manufacturing	Services
United States	- 7.8	- 0.5	+0.2	+ 8.3
Canada	15.1	- 4.4	- 3.7	+ 19.5
France	- 11.8	+ 3.1	+ 1.5	+ 8.8
Federal Republic of Germany	- 13.1	+ 5.9	+6.7	+ 7.2
Italy	-25.1	+ 13.1	+ 8.5	+ 11.9
Japan	- 26.2	+ 12.1	+9.7	+ 14.1
United Kingdom	2.5	1.7	+0.3	+ 4.2
Sweden	14.8	- 0.2	-2.9	+ 15.0

SOURCE: Bureau of Labor Statistics, Washington, D.C.

facturing industries, automation and technological change has brought about a shift of manpower from manufacturing industries into services. See Table 5. Also, working-hour schedules are becoming less and less rigid and the work content physically less strenuous. The number of wage earners (paid per hour of work) will decrease continuously, whereas the number of employees (paid per month) will increase correspondingly. These in turn will contribute to the stabilization of the economy, thanks to more stable purchasing power.

Hierarchic management systems are no longer adapted to modern production techniques, and computer utilization in management techniques make them obsolete. Participative management and motivated management methods are being increasingly used in industry and services. Training and retraining of personnel and permanent education programs permit the working force to keep abreast of technological knowledge.

Technical progress involves new methods of utilizing the labor force, especially as a result of the new quantitative, structural, and technical training requirements that workers must meet because of increasing automation. These labor-force plans must provide for the following requirements:

1. Meeting labor-force requirements in relation to the economic changes in the branches of industry (short-term adjustments).
2. Determining the change in the prospective structure of the labor force on the basis of the present occupational pattern (long-term adjustments).
3. Determining the general prospective trend of the population and the demands made on the educational system (structural change).

Because of the universal nature of information processing, which is applicable to a great many of

the activities in industry and administration, broad instructional programs for dissemination of automation techniques must be incorporated into educational curricula.

The introduction and development of automation in developing countries can be expected to help overcome the handicaps created by the scarcity of skilled labor for industrial production, thus intensifying the pace of industrialization, widening and consolidating intersectoral relationships (agriculture, manufacturing, services), and increasing the rate of economic growth.

REFERENCES

1970. International Institute of Labour Studies. "Employment Problems of Automation and Advanced Technology-An International Perspective." Geneva, Switzerland.
1971. International Labour Organisation. "Automation: Some Classification and Measurement Problems." Geneva, Switzerland.
1971. United Nations. "Economic and Social Aspects of Automation." U.N. Economic Commission for Europe, Sales No. 4. E. 70. II. E. 16. Geneva, Switzerland.
1972. U.S. Dept. of Commerce. "U.S. Industrial Outlook 1972 with Projections to 1980." Washington, D.C.: U.S. Government Printing Office.

F. MULLER

AUXILIARY MEMORY. See **MEMORY:**
Auxiliary,

B

BCD. See **B**INARY **C**ODED **D**ECIMAL, **N**ATURAL.

BCS. See **B**RITISH **C**OMPUTER **S**OCIETY.

BABBAGE, CHARLES

For articles on related subjects see **D**IGITAL **C**OMPUTERS: **H**istory; and **S**TORED **P**RO-**G**RAM **C**ONCEPT.

Charles Babbage was born in Totnes, Devonshire, on Dec. 26, 1791. He was educated privately and went up to Cambridge in 1810. At that time, Cambridge education was strongly oriented toward mathematics and there was intense competition for high honors in the Mathematical Tripos. Babbage, however, soon discovered that Newton's ideas still dominated the Cambridge curriculum, whereas he had been exposed to, and was much drawn toward, the type of mathematics then receiving attention on the Continent. He did not, therefore, compete for honors. Nevertheless, he acquired a high mathemat-

ical reputation which increased with the years, so much so that in 1828 he was appointed **L**ucasian Professor, a position that Newton himself had held many years before. The stipend in Babbage's time



Fig. 1. Charles Babbage. *Courtesy of New York Public Library .*

BABBAGE, CHARLES

was only £80 to £90 per annum. He did not reside in Cambridge nor lecture there, though he performed some of the other duties of the Professorship, such as examining for the Smith's Prize.

It was while still a student that Babbage began to work on the difference engine, a device intended to mechanize the production of the final values in a mathematical table from the widely spaced pivotal values that are first computed. It would also produce a stereotype mold, ready for the printer, thus eliminating one source of error. Babbage's own attempt at implementing the difference engine failed, in spite of financial support from the British government. The soundness of his ideas was, however, demonstrated by the fact that an independent implementation by George and Edward Sheutz, who had read of Babbage's ideas, was successful.

In a long life, Babbage turned his attention to many subjects, including mathematics, railroads, lighthouses, economics, the ophthalmoscope, politics, and public controversies of various kinds. But the dominating interest of his life was calculating machinery, and his claim to fame is through his work on the analytical engine, which was to have been an automatically sequenced, general-purpose calculating machine. Here he was profoundly original. He published some of his ideas and others have come down to us in his manuscript notebooks. The real breakthrough came in 1834 and the years immediately following, but Babbage continued to work on the subject for the remainder of his life.

Babbage's thoughts on the analytical engine were entirely in mechanical terms, with no suggestion, even in later years, that electricity might be called in aid. The analytical engine was to be decimal, although Babbage considered other scales of notation. Numbers were to be stored on wheels, with ten distinguishable positions, and transferred by a system of racks to a central mill, or processor, where all arithmetic would be performed. He had in mind a storage capacity for a thousand numbers of 50 decimal digits. He studied exhaustively a wide variety of schemes for performing the four operations of arithmetic and he invented the idea of anticipatory carry, which is much faster than carrying successively from one stage to another. He also knew about hoarding carry, by which a whole series of additions could be performed with one carrying operation at the end.

The sequencing of the analytical engine was to have been fully automatic, but not, of course, on what we would now call the stored-program principle. Punched cards of the type used in a Jacquard loom were to be adopted both for sequencing and

for the input of numbers. Babbage proposed to have two sets of sequencing cards, one for controlling the mill and one for controlling the store; these would be separately stepped and would not necessarily move together.

Babbage never arrived at the idea of instructions containing both an operation part and an address part, nor at the formal concept of a program that we have today. Lady Lovelace, the daughter of Lord Byron, in notes to a translation that she made of a paper describing some of Babbage's ideas, published by **Ménabréa** in French in 1842, gives what at first sight appears to be a program along modern lines for computing Bernoulli numbers. This gives the arithmetic operations in detail, but does not contain anything corresponding to the conditional jump instructions in a modern program; after the main loop there is simply the sentence: "Here follows a repetition of operations 13 to 23."

Babbage's notebooks show him struggling with various ideas for handling the repetition of parts of a calculation, and although he sketched out many schemes that would have worked satisfactorily, one feels that he never arrived at one that entirely pleased him. For subsequencing within an operation he proposed to use drums with fixed studs, on the barrel-organ principle. It is odd that in his published writings there is no hint of the range and originality of his thoughts on the important matter of sequencing. Lady **Lovelace** has left us in her debt for the translation and notes referred to above, but there has been a tendency to exaggerate her importance in the Babbage saga.

Although he had workmen in his employ until the end of his life, Babbage failed to implement the analytical engine. We must conclude, as did some of his contemporaries, that he was temperamentally incapable of carrying a project through. Unfortunately, this time, there was no Sheutz to take up his ideas, and it may well be that the ultimate development of automatic calculating machinery was delayed by the aura of failure that surrounded Babbage's work. His detailed design studies lay buried in his unpublished notebooks and were forgotten. Of his genius, however, no one who has studied his work will have any doubt.

Babbage died in London on Oct. 18, 1871. His youngest son, Henry, who had spent most of his life in various military and civil appointments in India, did what he could to carry on his father's work, and published a collection of papers relating to it. The eldest son, Herschel, migrated in 1851 to South Australia, where he became a prominent member of the colony.

REFERENCES

1889. Babbage, H. P. (Ed.). *Babbage's Calculating Engines*. London.
1961. Morrison, P., and E. Morrison (Eds.). *Charles Babbage and His Calculating Engines*. New York: Dover.
1968. Babbage, C. *Passages from the Life of a Philosopher*. London, 1864; facsimile edition, London: Dawson's.
1971. Wilkes, M. V. "Babbage as a Computer Pioneer," Report of the Babbage Memorial Meeting, British Computer Society.

M. V. WILKES

BACKTRACKING

For articles on related subjects see ALGORITHM; and HEURISTICS.

Often problems occur for which the only possible (or known) method of solution is by a search through the space of (perhaps a very large number) all possible solutions. In such cases it is important to attempt possible solutions systematically, eliminating potential solutions as quickly as possible, and never retrying a potential solution that has already been **tried**. The backtracking technique achieves these aims.

Two examples will suffice to explain backtracking. Fig. 1 shows a simple maze. Our systematic technique to find a path through the maze is to always keep to the right until a dead end is reached, then **backtrack** until an alternate path is found, again keep to the right, etc. The dark line shows the path through the maze, the light lines show the other paths traversed in finding this path.

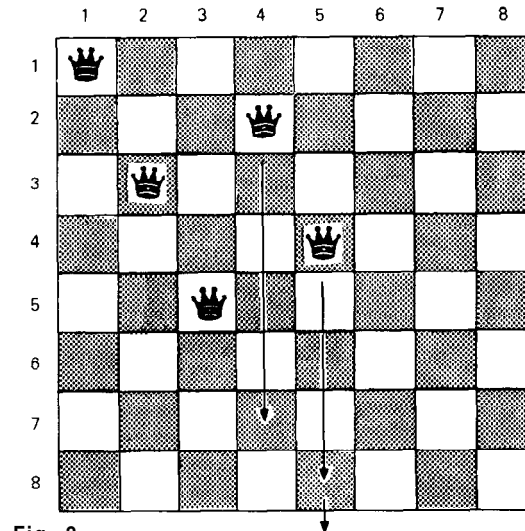


Fig. 2

Fig. 2 shows a chessboard at a stage in an attempt to solve the 8-Queen's problem in which eight queens are to be placed on a chessboard so that no two are in the same row or column or on the same diagonal. Our technique begins by placing a queen in the first column on the first row (1,1). The next queen is then placed in the second column on the first possible square, the third (3,2). Then subsequent queens are placed on (5,3), (2,4), and (4,5) as shown in Fig. 2. But with this arrangement no queen can be placed in column 6, so we **backtrack** to column 5 and move the queen to (5,8), again try but fail to place a queen in column 6, backtrack to column 5 but then, since we are already in row 8, backtrack again to column 4, move the queen on (2,4) to (7,4), etc. Eventually we find the first of 92 solutions, 12 of which are really distinct, that is, not related to others by some type of symmetry, at (1,1), (5,2), (8,3), (6,4), (3,5), (7,6), (2,7), (4,8). This method of solving the 8-Queen's problem can be easily and efficiently programmed for a computer (see Wirth, 1971).

Other areas in which backtracking is a commonly employed and useful technique are in the

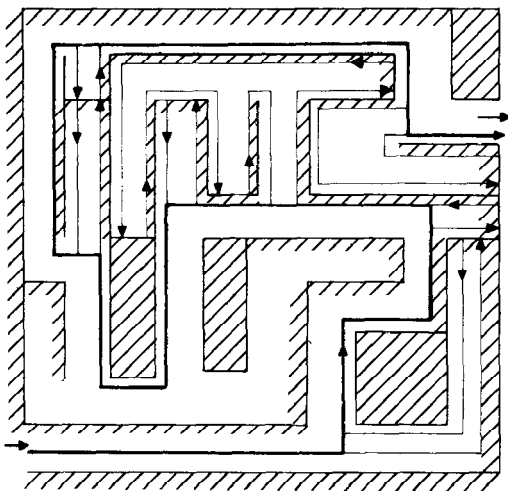


Fig. 1

BACKUS-NAUR FORM

parsing process in the compilation of a higher level programming language, theorem-proving and game-playing programs, and generally those tasks whose search domain is "ill structured." A formal description of backtracking is found in Golomb and Baumert (1965).

REFERENCES

1965. Golomb, S. W., and L. D. Baumert. "Back-track Programming," *Journal of the ACM*, Vol. 12, pp. 517-524.
1971. Wirth, N. "Program Development by Stepwise Refinement," *Communications of the ACM*, Vol. 14, pp. 221-227.
- 1973, ———, *Systematic Programming: An Introduction*. Englewood Cliffs, N. J. : Prentice-Hall.

A. RALSTON

BACKUS-NAUR FORM

For articles on related subjects see **META-LANGUAGE**; **PROCEDURE-ORIENTED LANGUAGES**, Survey of; and **PROGRAMMING LANGUAGES**.

For articles on related terms see **RECURSION**; and **SYNTAX**, **SEMANTICS** AND **PRAGMATICS**.

The Backus-Naur form, named after John W. Backus of the United States and Peter Naur of Denmark, and usually written BNF, is the best-known example of a *metalanguage*; i.e., one that syntactically describes a programming language. Using BNF it is possible to specify which sequences of symbols constitute a syntactically valid program in a given language. (The question of *semantics*-i.e., what such valid strings of symbols mean-must be specified separately.) A discussion of the basic concepts of BNF follows.

A *metalinguistic variable* (or metavariable), also called a *syntactic unit*, is one whose values are strings of symbols chosen from among the symbols permitted in the given language. Metalinguistic variables are enclosed in brackets (()) for clarity and to distinguish them from symbols in the language itself. The symbol $::=$ is used to indicate metalinguistic equivalence; a vertical bar (|) is used to indicate that a choice is to be made among the items so indicated; and concatenation (linking together

in a series) is indicated simply by juxtaposing the elements to be concatenated.

For an example, here is how the definition of an Algol integer is built up; First, we have a definition of what a digit is, according to the usual meaning:

(digit) $::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$.

Next we have a statement that an unsigned integer consists either of a single digit or an unsigned integer followed by another digit:

(unsigned integer) $::=$ (digit) |
(unsigned integer) (digit)

This definition may be applied *recursively* to build up unsigned integers of any length whatever. If there were to be a limit on the number of digits, it would have to be stated separately in conjunction with each particular implementation. (The Algol language contains no such limitation, but all computer implementations must, of course, have some limitation.) Finally, the definition of an integer is completed by noting that it may be preceded by a plus sign, a minus sign, or neither:

(integer) $::=$ (unsigned integer) I +
(unsigned integer) | - (unsigned integer)

For a second example, suppose that the metalinguistic variables (unsigned number), (variable), (function designator), and (boolean expression) have all been defined earlier, with usual meanings, and that the up-pointing arrow stands for exponentiation. Here, then, is the complete definition of an Algol arithmetic expression:

(adding operator) $::= + | -$
(multiplying operator) $::= \times | / | \div$
(primary) $::=$ (unsigned number) | (variable) |
(function designator) | ((arithmetic expression))
<factor> $::=$ (primary) I (factor) \uparrow (primary)
(term) $::=$ (factor) I (term) (multiplying operator) (factor)
(simple arithmetic expression) $::=$ (term) I
(adding operator) (term) |
(simple arithmetic expression) (adding operator) (term)
(if clause) $::=$ if (boolean expression) then
(arithmetic expression) $::=$ (simple arithmetic expression) I
(if clause) (simple arithmetic expression)
else (arithmetic expression)

It is no error that the third definition contains (arithmetic expression), enclosed in parentheses, even though it is an arithmetic expression that we are trying to define. This also is a matter of recursive definition, and simply says in this case that one choice for a (primary) is just any (arithmetic expression) enclosed in parentheses.

The words **if**, **then**, and **else**, since they are not enclosed in the **meta-linguistic** brackets, stand for themselves; they are elements of the Algol language and are defined elsewhere in the complete Algol language definition.

Almost any programming language can be defined in BNF, but somewhat different **meta** languages have generally been used with the major procedure-oriented languages other than Algol.

REFERENCE

1969. Sammet, Jean E. *Programming Languages: History and Fundamentals*. Englewood Cliffs, NJ: Prentice-Hall (Chap. 2, sec. 6).

D. D. McCracken

BANDWIDTH

For articles on related subjects see **COMMUNICATIONS AND COMPUTERS**; and **DATA COMMUNICATIONS**.

The bandwidth of a communication network is a measure of the range of frequencies it can transmit at or near maximum power levels. As an example, consider a normal telephone system, which is an analog communication network normally designed to carry voice traffic in the frequency range 300-3400 Hz. Thus the equipment in the telephone exchange collects incoming data from the sound spectrum and arranges to attenuate sharply the signals outside that part of the spectrum. But even within that range there is further attenuation as the signals propagate through the telephone network, since the power of signals passing through the telephone transmission system is reduced. A typical measurement on the U.S. telephone network of attenuation is shown in Fig. 1, which indicates that somewhere below 300 Hz and above 3-4K Hz, the attenuation rises very rapidly. The range of frequencies in which the power level stays at above one-half its peak value (the so-called 3 db points) is

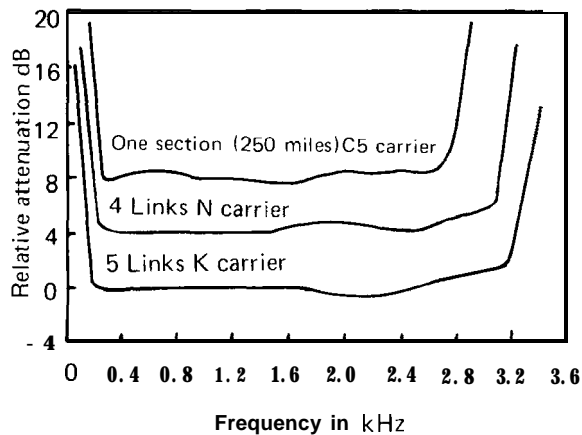


Fig. 1. Attenuation for frequency division multiplexing (FDM) systems. (reproduced from *Communication-Networks for Computers* by D. W. Davies and D. L. A. Barber. New York: Wiley, 1973, Fig. 2.16.)

the *nominal bandwidth* of the circuit. This is typically 3K Hz in a switched telephone line.

P. T. KIRSTEIN

BASE REGISTER

For article on related subject see **ADDRESSING**.

For article on related term see **IBM 360-370 SERIES**.

A base register is used in addressing a computer memory. In a computer that uses base registers, the effective address (i.e., the address field of the instruction, possibly modified by indexing and indirect addressing) is a relative address. The actual memory address used is determined by adding this relative address to the contents of one or more base registers.

The Control Data Cyber 70 series is an example of a computer system that uses a single base register. Every program is written as if it were meant to run in a single memory area, starting at location 0. The program may in fact be loaded starting at any memory location. When the program is run, the operating system places the address of the first word of the program in the base register. The content of the base register is automatically added to every memory reference address, and thus every relative

BASIC

address is converted into an absolute address. This feature is useful in multiprogramming systems, since it permits programs to be loaded wherever space exists, and permits programs to be moved in memory, or to be removed from memory and then resumed in a different area of memory. Such base registers are thus often called “relocation registers,”

Some computers have several base registers. A relative address must then contain a field that indicates which register is selected, and the contents of that register are added to the relative address to form the absolute address. In such a machine a program may be constructed in parts or segments that can be independently loaded into available areas of memory. The UNIVAC 1108 is an example of a machine with two base registers. The Multics machine (Honeywell GE645) is an example of a machine with four base registers.

The term “base register” is sometimes used more or less interchangeably with the term “index

register • ” Thus, the IBM 360 and 370 have 16 general registers, each of which provides a 24-bit base address to which the 12-bit address field (displacement) in an instruction is added to produce the effective address. These registers can be, and usually are, loaded by and stored in the programs that use them. It is conceptually better to think of them as index registers and to limit the use of the term “base register” to system registers that are not accessible to the programs whose addresses they modify.

S. ROSEN

BASIC. See PROCEDURE-ORIENTED LANGUAGES,

BAUD

For articles on related subjects see BAUDOT CODE; and CHANNEL.

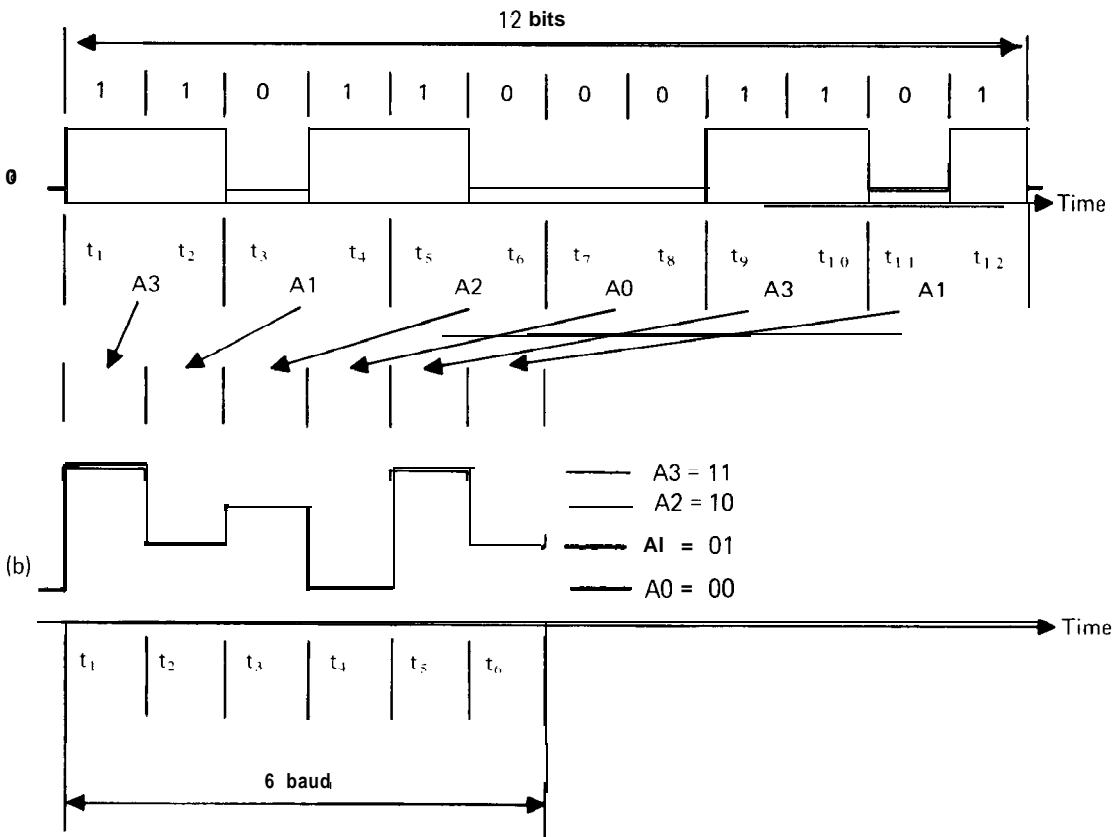


Fig. 1. Relationship between baud and bits per second. Each combination of two bits is encoded as one of four possible amplitudes, hence, one baud is equal to two bits per second.

A **baud** is a unit of signaling speed and refers to the number of times the state (or condition) of a line changes per second. It is the reciprocal of the length (in seconds) of the shortest element in the signaling code. Historically, it is a contraction of the surname of the Frenchman J. M. E. Baudot, whose five-bit code was adopted by the French telegraph system in 1877. By contrast, a bit is the smallest unit of **information** in a binary system. The baud rate is therefore equal to the bit rate only if each signal element **represents** one bit of information.

The relationship between bauds and bits per second is illustrated in Fig. 1, where amplitude is used as a coding method. In this particular case, there are four line conditions, one for each of the four combinations of two bits. Each line change signal element is therefore represented by two bits and, if we can have one line change in 1 ms, the baud rate is 1,000, whereas the bit rate is actually 2,000 bits per second. Similarly, if the signals are coded into eight possible states, one line condition could represent three bits and one baud would then equal three bits per second and so on.

Unfortunately, in much of today's literature, the terms "baud" and "bits per second" are used synonymously. This is correct in cases where pure two-state signaling is used, as in Fig. 1(a), but is incorrect in general. For this reason, the term "baud" is gradually being replaced by "bits per second," since the latter is independent of the coding method and truly represents the information rate.

J. S. SOBOLEWSKI

BAUDOT CODE

For articles on related subjects see **BAUD**; **CODES**; and **ERROR CORRECTING CODE**.

The Baudot code, also known as the **International Telegraph Code No. 1**, is named after its inventor, J. M. E. Baudot (1845-1903). It was invented about 1880, and by the 1950s it had become one of the standards for international telegraph communication.

Baudot is a fixed character-length code in which each character is represented by five binary digits. The **5-digit** character length allows for only 32 unique combinations, not enough to represent the 26 letters of the alphabet, the 10 digits, and the **punc-**

uation characters needed for telegraph messages. This problem of digit limitation is solved by defining two unique shift-control characters, and interpreting all subsequent characters in terms of the last **shift-control** character received. The shift-control **characters** are called "letter shift" and "figure shift". This arrangement is very similar to that of a shift-lock key on a typewriter, i.e., once the shift lock has been depressed, all subsequent characters are typed in the same shift.

Using the technique of two unique shift characters, a **5-bit** code can then represent 62 characters. However, in the Baudot code the total number of characters is less than this because other control characters such as "line feed" and "carriage return" are given unique representations.

Table 1. Baudot Code Characters.

<i>Letters</i>		<i>Figures</i>
A	10 000	
B	00 110	8
c	10 110	9
D	1 1 1 1 0	Ø
E	0 1 0 0 0	2
F	0 1 1 1 0	NA
G	0 1 0 1 0	
H	11010	+
I	01 100	NA
J	10 010	6
K	10 011	(
L	11011	=
M	0 1 0 1 1)
N	0 1 111	NA
O	1 1 1 0 0	5
P	1 1 1 1 1	%
Q	10 111	/
R	0 0 1 1 1	-
S	00 101	.
T	10 101	NA
U	10 100	4
V	11 101	'
W	01 101	?
X	01001	,
Y	00 100	3
Z	11001	:
LS	0 0 0 0 1	LS
FS	0 0 0 1 0	FS
CR	11 000	CR
LF	1 0 0 0 1	LF
ER	0 0 0 1 1	ER
NA	00 000	NA

Symbols: LS = Letter Shift, FS = Figure Shift, CR = Carriage Return, LF = Line Feed, ER = Error, NA = Not Assigned, Space = LS or FS.

BENCHMARK

The Baudot code does not have the capability of detecting errors because all combinations of the five bits are valid characters within the code. During transmission, therefore, a character can be transformed into **another** character by the loss or gain of one or more bits. Particularly harmful is an error in a shift-control character because all characters after the transformed shift-control character up to the next shift-control character would be interpreted in the wrong shift. For example, in the message **PAY 8 10 DOLLARS**, if the "figure shift" character between the **PAY** and the **810** were transformed into, say, a **J** (i.e., 00010 to 10010), then the message would be received as **PAYJBAD DOLLARS** (see Table 1 for letter-shift, figure-shift equivalents). In order to alleviate this problem, telegraph systems frequently retransmit at the end of the message all figures that occur in the message.

The five-level code most used today is the International Telegraph Code No. 2 (Murray code), invented about 20 years after the Baudot code. In computer manufacturer's literature, there is some confusion concerning the use of the term "baudot code." It is sometimes used to apply to all five-level codes and is frequently applied to International Telegraph Code No. 2.

G. D. DETLEFSEN AND R. H. KERR

BENCHMARK

For articles on related subjects see **GROSCH'S LAW**; and **PERFORMANCE OF COMPUTERS**.

Benchmarks are standardized computer programs used to test the processing power of different computers. They specify the input data, the computations to be performed, and the output formats very rigidly while leaving the details as to how the calculation is to be performed flexible enough to allow the individual advantages of a particular computer being tested to be maximized. (See Sharpe, 1969.)

Auerbach Information, Inc., who have been providing benchmark comparisons since 1962 in their "Standard EDP Reports," uses five general benchmark programs to compare computers: (1) a generalized file processing problem, (2) a random access file processing problem, (3) a sorting problem, (4) a matrix inversion problem, and (5) a generalized

mathematical problem. This is probably the longest running series of benchmark programs and allows general comparison between machines over a considerable time frame.

If the user has a specific type of problem that will account for a large percentage of the use of his machine, he can run more specific benchmark problems to evaluate various manufacturers' products.

Benchmark programs are one way by which machine characteristics can be compared. Alternatively, a user may specify a set of instructions and compare machines on the basis of how well they perform this instruction mix. This allows comparison among machines regardless of programming language, hardware construction, etc.

Benchmark programs may either be actually run or their performance may be calculated from the manufacturer's published data on the characteristics of the computer. Either form of comparison may be useful in evaluating computers.

Benchmarks are used by computer purchasers to determine what machine is best for their particular use in terms of both speed and cost. However, benchmarks alone cannot be used for this purpose, since most large, general-purpose computers will be used to handle a wide class of problems whose frequency will be unpredictable before purchase. Thus, in addition to benchmarks, other evaluations, subjective and objective, will enter into the decision of what computer to obtain.

One example of another objective comparison is the computer power versus cost formula developed by Knight and Cervený in the article *Performance of Computers*, which allows very general comparisons both by power/cost and by year.

REFERENCE

1969. Sharpe, W. F. *The Economics Of Computers*. New York: Columbia University Press, pp. 308-3 12.

K. E. KNIGHT AND R. P. CERVENY

BINARY CODED DECIMAL, NATURAL

For article on related subject see **CODES**.

Natural binary-coded decimal (NB CD) is a particular binary-coded decimal (BCD) code that is

itself often called **BCD**. It is "natural" because it uses the first ten binary numbers in sequence to represent the digits 0 through 9 (see Table 1).

Table 1. NBCD Code.

Digit	NBCD Combination
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Since 9 is represented by 1001, NBCD is a four-bit code. Moreover, it is a weighted code, since the four bits, left to right, correspond to weights of 8, 4, 2, and 1. Thus, 0110 represents a decimal 6.

Six possible four-bit combinations (1010, 1011, 1100, 1101, 1110, 1111) are not part of the code and are called "forbidden" combinations.

I. FLORES

BINARY SEARCH

For articles on related subjects see **COLLATING SEQUENCE**; **KEY**; **SORTING**; and **TABLE LOOKUP**.

Binary search is a quick method for searching an ordered, dense list (i.e., every cell of the list contains a record) for a particular record by successively looking at one half of the list in which the record is known to be.

A record is recognized by a field, called its "key." We assume that the list is ordered with respect to the collating sequence of its keys.

There are several ways to find a specific record identified only by its key. The simplest, but longest, is the sequential search, in which cells of the list are searched in the physical order as they appear in the list. Binary search considerably reduces the time required to find a desired record.

To present the basic idea of binary search, we introduce the idea of a "fence cell" or simply a fence that divides a list into two equal or nearly equal parts. Examining the key of the fence record will tell

us which half of the list contains the desired record. A fence is set up for the half-list that contains the record and then the quarter-list that contains the record is determined. From the quarter-list we determine an eighth-list and so forth. The actions of binary search then consist of three phases:

1. Dividing a list or **sublist** into two fairly equal parts by defining a fence.
2. Examining the fence to determine which half to use next.
3. Determining when our search is finished.

To present the details of the mathematics of binary search, we define the parameters:

S = address of the starting point of the list in memory.

iv = length of the list in number of cells.

c = size of each cell in bytes or words, or other appropriate units.

K = key of the record sought.

f_i = ordinal number in the list of the i th fence.

F_i = address of the i th fence.

n_i = number of items in the i th **sublist** with $n_0 = N$.

To begin the binary search we calculate

$$f_1 = [n_0/2] + 1, \quad (1)$$

where the square brackets denote the integer less than or equal to the number contained within them. Then

$$F_1 = S + C(f_1 - 1). \quad (2)$$

To find F_2 , we must determine in which half of the list the desired record is contained. Denote by $(F_1)_K$ the key in cell F_1 . Then, either (a) $K = (F_1)_K$, in which case the search is finished; or (b) $K < (F_1)_K$, in which case the lower **sublist** is used and

$$n_1 = \left[\frac{n_0}{2} \right] \quad f_2 = f_1 - \left[\frac{n_1}{2} \right]; \quad (3)$$

or (c), $K > (F_1)_K$, in which case the upper sublist is used and

$$n_1 = n_0 - 1 - \left[\frac{n_0}{2} \right] \quad f_2 = f_1 + \left[\frac{n_1}{2} \right] + 1 \quad (4)$$

The difference in the equations for f_2 is to insure that the lower half-list is always the same length or one

BINDING TIME

greater in length than the upper half-list. In any case,

$$F_2 = S + C(f_2 - 1), \quad (5)$$

and then we proceed to find F_3, F_4, \dots by a similar procedure, using

$$\begin{aligned} f_{i+1} &= f_i - \left\lfloor \frac{n_i}{2} \right\rfloor \text{ or } f_i + \left\lfloor \frac{n_i}{2} \right\rfloor + 1, \\ F_{i+1} &= S + C(f_{i+1} - 1), \\ n_{i+1} &= \left\lfloor \frac{n_i}{2} \right\rfloor \text{ or } n_i - 1 - \left\lfloor \frac{n_i}{2} \right\rfloor. \end{aligned} \quad (6)$$

At some point K must equal $(F_i)_K$ and the search is finished; otherwise, the desired record is not in the list. The latter case is automatically discovered when $n_i = 1$ and the key does not match K .

Fig. 1 illustrates the binary search procedure for the case $N = 28$ and where the record sought is in cell 11. For convenience we assume $S = C = 1$. Using the foregoing equations, we have

$$\begin{aligned} f_1 &= F_1 = 15, & n_1 &= 14, \\ f_2 &= F_2 = 8, & n_2 &= 6, \\ f_3 &= F_3 = 12, & n_3 &= 3, \\ f_4 &= F_4 = 10, & n_4 &= 1, \\ f_5 &= F_5 = 11, & & \end{aligned}$$

at which point we have found the desired record.

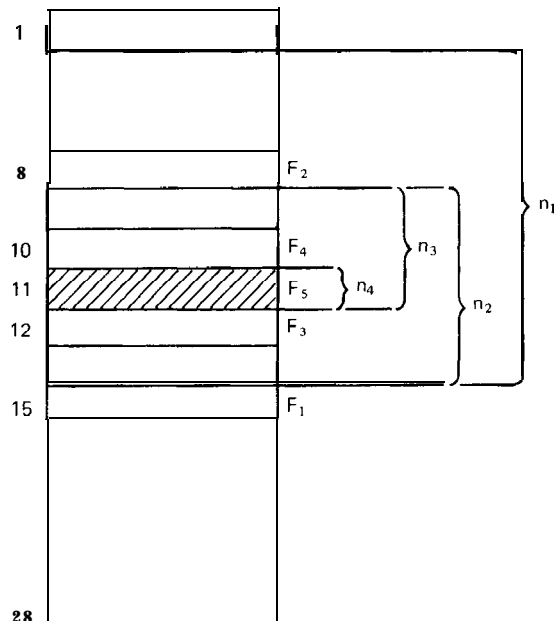


Fig. 1. Binary search.

The efficiency of a search procedure can be judged according to the number of "looks" it takes to find the desired record. In the case of binary search, the action proceeds by continuously dividing successive **sublists** in half. This is a logarithmic process, using 2 as a base. It is not hard to see that the largest number of looks required to find the desired item is given as

$$L = \lceil \log_2 N \rceil + 1. \quad (7)$$

In the preceding example, $\lceil \log_2 28 \rceil + 1 = 5$. Often it will not take this many looks because one of the fences that we examined along the way will actually contain the record sought. If we call \bar{L} the average number of looks, where L is the maximum number of looks, then this average number is such that

$$L - 1 < \bar{L} \leq L. \quad (8)$$

REFERENCE

1970. Flores, Ivan. *Data Structures and Management*. Englewood Cliffs, N.J. Prentice-Hall, pp. 7 1-75.

I. FLORES

BINDING TIME

For articles on related subjects see **ASSEMBLERS**; and **LANGUAGE PROCESSORS**.

Binding time refers to the instant when a symbolic expression in a computer program is reduced to a form directly interpretable by the hardware. For example, a symbolic expression given to a translator is progressively bound as that translator does its work. Normally binding time refers to that moment when the symbol or expression concerned has been reduced to a bit pattern in a fixed location in the storage device concerned. But binding time can also refer to an intermediate point in this process if the context in which it is used makes this clear. For example, the translator under discussion may leave certain symbols incompletely bound, letting a linking loader complete the job. Used in such a context, binding time can mean the point at which that translator has gone as far in binding the symbol as it can.

Binding time is often used with reference to the allocation of storage in higher level language programs. For example, in **Fortran** the binding time for all storage allocation is at compilation or loading time. In other languages like **Algol** or **PL/I**, data is dynamically created and erased while a program is executing so that the binding time for much storage allocation is at run time.

In general the later something is bound the more flexibility it provides for the programmer but the harder it is for the computer to deal with. Binding sometimes takes place progressively as happens when a language processor binds various parts of an expression at different stages of the translation process. The rate at which binding takes place, and the moment when it has taken place completely, are important design considerations for all sorts of software. Early binding means efficiency, with character strings replaced by addresses and expressions by binary numbers. For the same reasons, it means loss of debugging information and the ability to optimize by recognizing identical constants, common subexpressions and the like. Broadly speaking, the history of software development is the history of ever-later binding time, as user convenience is given more attention and efficiency left to faster hardware and specialized translators that are employed only after all debugging is done.

Some translators have attempted to put the question of binding time, to some degree, in the users' hands. One, at least (Strachey, 1968) has gone all the way for experimental purposes, letting the user specify the rate at which each of his symbols is to be bound.

REFERENCES

1968. Strachey, C. "A General Purpose Macro Generator," *The Computer Journal*, Vol. 8, pp. 225-241.
1968. Wegner, P. *Programming Languages, Information Structures and Machine Organization*. New York: McGraw-Hill Book Company.

M. HALPERN

BIOMEDICINE, COMPUTER GRAPHICS IN

For articles on related subjects see **COMPUTER GRAPHICS**; and **MEDICAL APPLICATIONS**.

For many years, investigators of complex biological models have welcomed interactive graphics. When their problems taxed conventional analog computers, they contrived various devices to extend their capabilities and were among the early developers and users of hybrid computers. Interaction and a graphics display were among design criteria for the LINC, whose development for biological laboratory computing was supported by the National Institutes of Health (NIH). Around 1965, Levinthal's rotating molecules drew attention to the power of major digital graphics (Levinthal, 1966). Soon several research centers extended the demonstration of this power to a variety of biomedical applications. Further applications, NIH-supported development of graphics systems software and low-cost hardware, and commercial advances have recently culminated in cost-effective graphics hardware and software resources that can and should advance rapidly throughout the biomedical domain.

This brief review is necessarily incomplete and will select projects that illustrate important categories of biomedical graphics applications.

Complex *three-dimensional structures (3-D)* abound in biology. The sensation of *three-dimensionality* and variety of perspectives provided by dynamic graphical presentations have been unprecedentedly effective in revealing atom-rich planes and suggesting active structures in biological molecules (Levinthal, 1966). Z-brightening, which increases the plotting intensity of portions of the structure "nearer" to the observer, enhances the three-dimensional sensation in viewing rotating displays. Interaction facilitates exploration of van der Waals limitations on rotations of molecular substructures, investigations of adding substituents, and a variety of visualizations from basic carbon skeletons to complete structures highlighting certain atoms. Valuable insights concerning embryonic evolution of the nervous system are being derived from graphically supported 3-D reconstructions from microscopic serial sections. Development of craniofacial structures and the mechanics of limb motion also are subjects of graphics-supported investigations.

Early expectations of full 'automation in diagnostic radiology and other areas of biomedical *image analysis* underestimated skills of the trained eye. Many projects foundered at the level of feature perception or location of regions of interest. Interactive graphics permits such obstacles to be bypassed. An observer approximately indicates chromosomes in overlapping spreads, leaving tasks of precise delineation, measurement, and classification

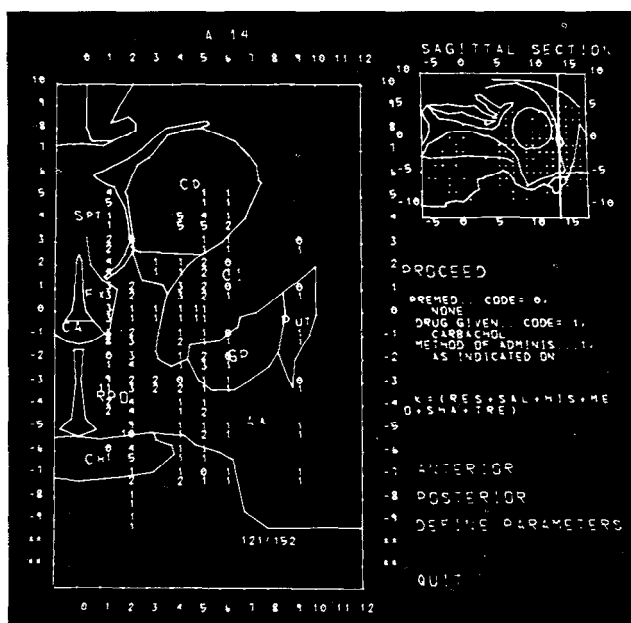


Fig. 1. Relating retrieved data to meaningful structures, A boolean combination of physiological and behavioral responses is related to sites of the brain where a given pharmacological agent has been injected. The notation $\frac{3}{4}$ means that the combination occurred three times, for four injections at the site.

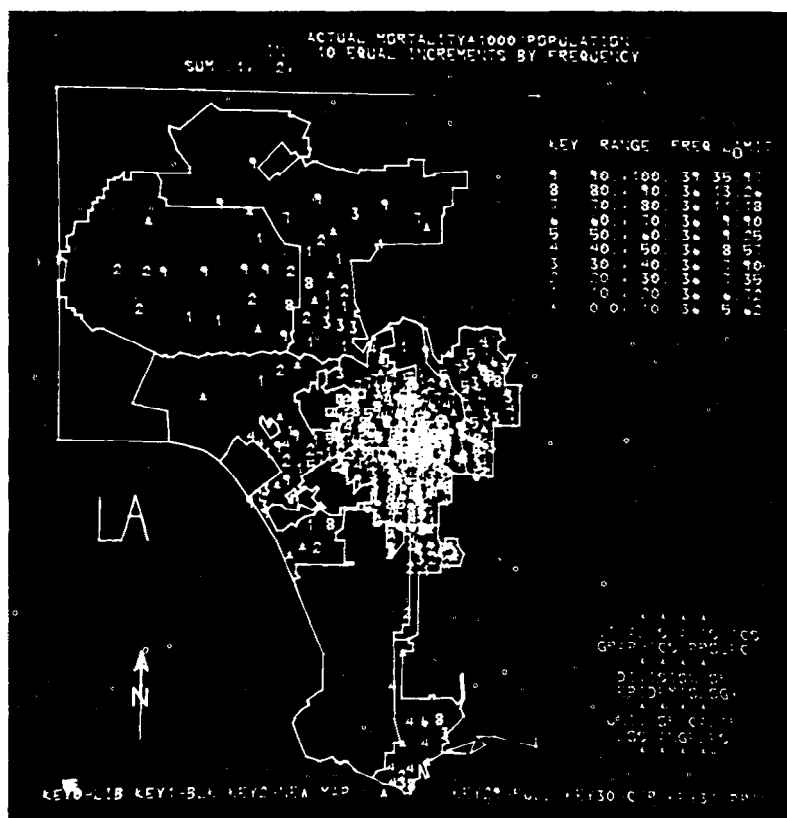


Fig. 2 (a).

to the machine (Neurath et al., 1966). The machine performs texture analysis for areas he indicates in a chest X-ray. The observer promptly reroutes when a rib or artifact entices edge tracking away from the heart in a sequence of radiographs used to study volume changes in normal and abnormal beating hearts. Gray-level video representations of the original X-ray films are back-projected onto the graphics screen for later application.

The power of interactive graphics finds full expression in *biological modeling* (Newton, 1972). Realistic mathematical description and productive investigation of difficult models encountered in biology and medicine demand high levels of analytical skill and biomedical knowledge. Few people

are trained to provide both, and even a variety of biomedical experts may be required for some problems. Properly designed graphics displays can communicate with these various scientists, providing further characterizations of underlying mathematical structures or biological assumptions as needed. Investigation of complex models is made tractable by techniques that monitor inconsistencies and otherwise facilitate model specification, as well as by meaningful multivariate displays during computation. Intuitive guidance of model exploration can result in substantial economies; the observer often can terminate a run well before the foreseeable stopping rules required in batch processing would take effect.

Fig. 2. (a) Demographic variables from the census tract can be associated with variables related to the incidence of heart disease in Los Angeles County. (b) User can zoom for closer study of neighborhoods.

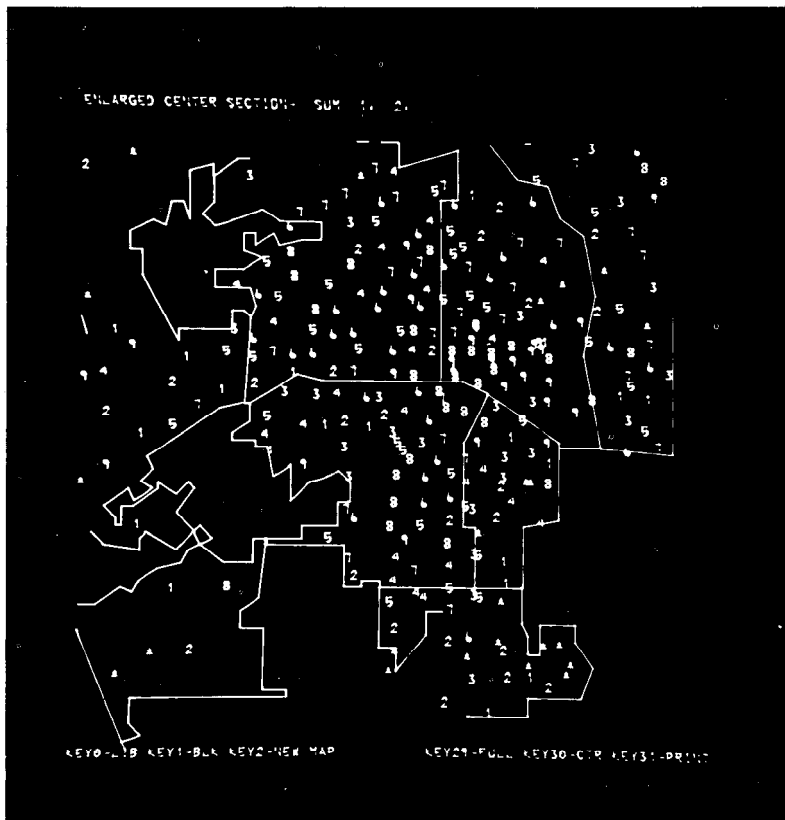


Fig. 2 (b).

Rand's **BIOMOD** system supports models that guide the treatment of leukemia and instruct medical students in fluid therapy (Groner et al., 1971). Others support exploration of multiagent strategies in cancer therapy and of realistic epidemiological models that incorporate timely features such as school busing and regional disparities in housing. After clarifying how diseases such as emphysema cause misinterpretation of certain conventional pulmonary function tests, a City of Hope internist directed his graphics model to the development of improved tests. Simpler graphics models of metabolic compartment analysis, enzyme kinetics, and community health care aid instruction.

Data examination in a rapidly evolving science is apt to emphasize hypothesis discovery; both physiological signals and data bases encountered in clinical studies are characteristically complex and noisy. Thus, the need to infuse human intuition and pattern perception into the analysis of biomedical data has motivated the development of a variety of interactive graphics programs for data exploration. Statistical programs range from novel approaches to profile analysis, to interactive graphics implementation of conventional time-series, regression, or discriminant function analyses. To specify retrieval, **lightpen** selection from branching menus or composition of boolean conditions from lists of descriptors provide an effective natural-language interface to biomedical investigators. Most important are meaningful displays of the retrieved information. The relationship of pharmacological action to anatomical neural pathways has been facilitated by research data summaries superimposed on a three-dimensional brain map, the sites of injection of pharmacological agents being known (Sheu et al., (1969) (see Fig. 1). The Los Angeles Heart Study data have been examined in conjunction with census demographic data, superimposed on a map of Los Angeles County, displayed in entirety or zoomed to any area (see Fig. 2.).

Interactive graphics data analysis has been directed to *designing economical patient-monitoring systems*. Alternative digital processing algorithms or simulated analog circuit designs operate upon physiological signals recorded from a large number of patients. The statistician/engineer and physician join in determining which works best and in hypothesizing how it might be improved. A cost-effective digital algorithm for monitoring electrocardiograms of coronary patients has resulted, as has inexpensive bedside analog circuitry for obstetrical monitoring.

Two additional examples of direct *medical applications* should be noted: In radiation treatment

planning, sources of radiation and interposed materials must be deployed to maximize dose delivered throughout the tumor and to minimize its effect on critical organs. Computation of dose distribution is complicated by tissue inhomogeneities, such as bone and air-filled lungs. The Programmed Console, an early graphics minicomputer turnkey system for exploring treatment alternatives, was an immediate success (Holmes, 1970). More versatile and powerful supports now are provided by remote graphics terminals that access major computers by dial-up telephone lines. Low-cost graphics would be welcomed to make more readable displays that superimpose the patient's anatomical diagram, dose-distribution contour plots, and related numerical data.

The computer itself is therapist in Colby's approach to treating nonspeaking autistic children, who reject humans but tend to be fascinated by machines (Smith et al., 1972). As they operate his responsive graphics-audio system, their progress toward speech is catalyzed to the extent that most soon can be hopefully referred back to previously unsuccessful conventional therapy.

REFERENCES

1966. Levinthal, C. "Molecular Model-Building by Computer," *Scientific American*, Vol. 2 14:42.
1966. Neurath, P. W., et al. "Human Chromosome Analysis by Computer-An Optical Pattern Recognition Problem," *Ann. N. Y. Acad. Sci.*, Vol 128, pp. 1013-1028.
1969. Sheu, Y., et al. "Topographic Information Retrieval in Neuropharmacology by Using Graphic Display," *Proc. Of 24th Annual Conference of the Assoc. Comput. Mach.*, pp. 485-489.
1970. Holmes, W. F. "External Beam Treatment-Planning with the Programmed Console," *Radiology*, Vol. 94, pp. 391-400.
1971. Groner, G. F., et al. *BIOMOD-An Interactive Computer Graphics System for Modeling*. The Rand Corporation, R-6 17-NIH (July).
1972. Newton, C. M. "Planning Radiotherapeutic Strategy," *Proc. San Diego Biomedical Symposium*, Vol. 11, pp. 189-203.
1972. Smith, D. C., et al. "Automated Therapy for Non-Speaking Autistic Children," *Proc. Spring Joint Computer Conference*, Vol. 40, pp. 1 101-1106.

C. M. NEWTON

3BLOCK AND BLOCKING

For article on related subject see **MEMORY: Auxiliary**.

For articles on related terms see **CHANNEL;** and **DATA STRUCTURES.**

The term "block" is synonymous with "physical record": a sequence of words or characters written contiguously by a computer on an external storage medium. Typically, one block is written each time a **WRITE** command is executed by an I/O channel (or equivalent I/O facility). Analogously, one block is read from an external medium each time that a **READ** command is executed by the channel.

"Block" is distinguished from "logical record" as follows: A block is defined by the physical characteristics and constraints of the external storage medium, whereas a logical record is defined by a particular data structure in a processing program. Logical records (often shortened to "records," al-

though this term is also loosely used for "blocks") are aggregates of data such as bits, numbers, and character strings, which are naturally and conveniently transmitted at one time from the main storage of a computer to an external medium. One type of data aggregate is a "master record," comprising all attributes associated with a member of some population.

Blocks typically contain several logical records when written onto magnetic media such as drums, tape drives, and disk drives. The size of a block is chosen to take into account the software characteristics of a system (e.g., buffer size) and the hardware characteristics of the external medium (so as to avoid too much starting or stopping when reading or writing tape). In the case of fixed-length blocks, the standard format is as shown in Fig. 1. For fixed-length logical records, the number of records per block is called the "blocking factor." For many computers (e.g., current IBM models), variable-length logical records are formatted into blocks, as shown in Fig. 2. Fig. 3 shows how large logical

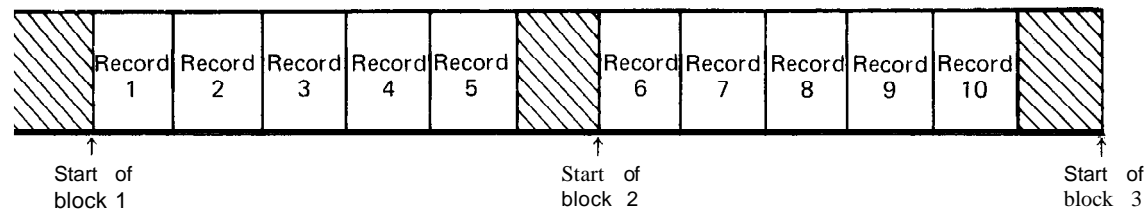


Fig. 1. Fixed-length blocks: blocking factor = 5; all records have same length; block length = $5 \times (\text{record length})$.

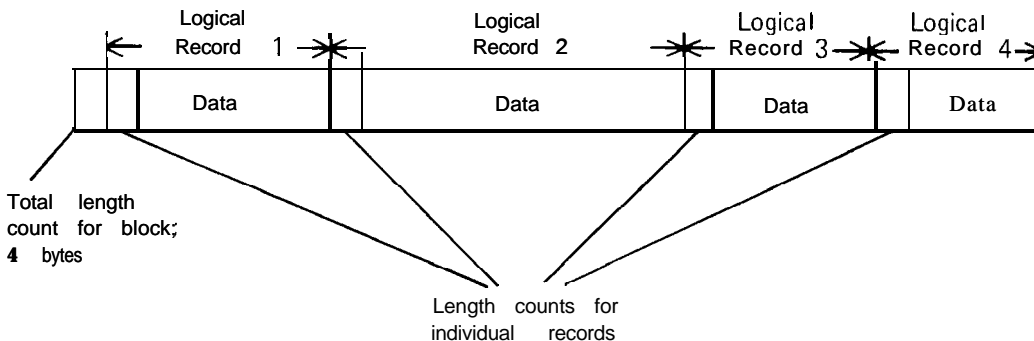


Fig. 2. Variable-block lengths (no logical record larger than one physical block): blocking factor, variable; block length = $4 + [(\text{length of record-1}) + 4] + [(\text{length of record-2}) + 4] + \dots$

BLOCK DIAGRAM

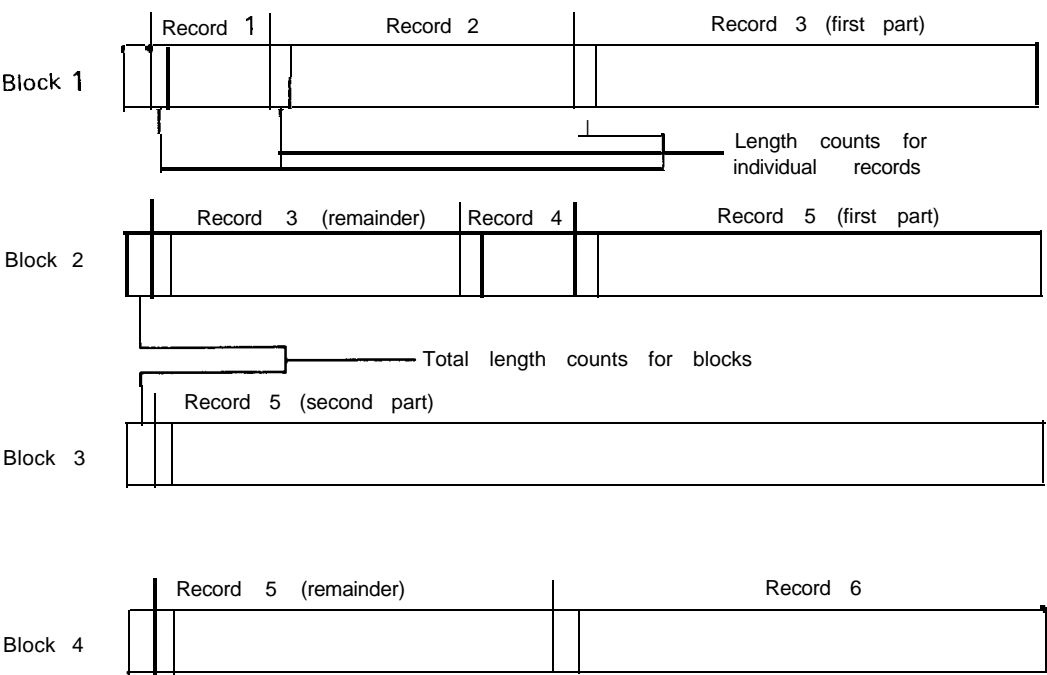


Fig. 3. Variable-length blocks (logical records may be larger than physical blocks).

records can be built up from smaller-sized blocks to obtain the “spanned record” format.

D.N. FREEMAN

Block diagrams may be drawn at any level of detail, but are most common at the grosser and more summary levels. Whatever the level of detail chosen, the usual focus of the block diagram is the control and data connections among the components identified. When a detailed representation of the interactions among components or circuit elements is

BLOCK DIAGRAM

For articles on related subjects see **FLOW-CHART**; **FLOW DIAGRAM**; and **SYSTEM CHART**.

A block diagram is a graphic means of representing the functions or components of a system in order to show the control or data connections among them. An example of a block diagram for a computer is given in Fig. 1.

In a block diagram, the components are normally labeled, but no attempt is made to present them either pictorially or dimensionally. Simple rectangles or circles are common. Lines or arrows generally indicate the operational directions of the control or data connections.

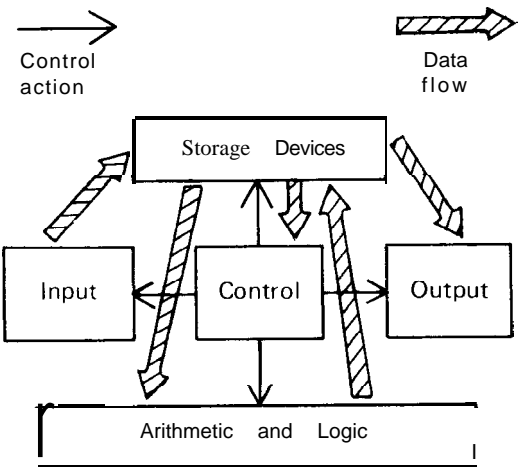


Fig. 1. Block diagram of a computer.

needed, the usual practice is to draw logic diagrams, functional diagrams, or circuit diagrams instead of block diagrams.

In the early days of the computer field, the term “block diagram” was also used as a synonym for flowchart. That practice is rarely followed today now that flowcharts have become more specialized in function.

N. CHAPIN

BLOCK STRUCTURE

For articles on related subjects see **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.
For articles on related terms see **PROCEDURE**; **SIDE EFFECT**; and **STACK**.

Block structure is one of the most powerful conceptual tools employed by the programmer. When used judiciously it can help transform a large, unwieldy program into a disciplined, well-structured, and easy-to-understand piece of code.
Because of the important functions it performs, block structure (first introduced in Algol 60) is found in one guise or another in practically all procedural languages developed after 1960.
After a brief look at the functions performed by block structure, from the programmer’s perspective, we take a careful look at the syntax, semantics, and run-time implementations of the Algol 60 block structure.

The Programmer’s View of Block Structure. From the programmer’s point of view, block structure performs two major control functions:

- 1. It groups a sequence of statements into a single “compound” statement.
- 2. It allows explicit control by the programmer over the scope of the variables he uses.

Groupings imply two things: First, we can reference a compound statement and use it wherever a single statement can be used (e.g., as the object of a loop). Therefore, we can think of a sequence of statements as a single entity and thus simplify the process of program construction. Second, we can decompose any large, unwieldy program into a disciplined nest of blocks, since a block may contain

other blocks as components. This is perhaps the most important use of the block from the programmer’s angle, for it allows him to construct his program in hierarchical fashion, which often results in increased program clarity and elegance.
Scope of variables implies three things: First, the ability to control dynamically (i.e., at run time) the allocation and freeing of storage. In particular, since the dimensions of all arrays, declared within the block are determined upon entry into a block, they may vary from invocation to invocation; second, the creation of a current context (environment), independent of all previous contexts, upon entry to a block, since locally allocated storage is freed upon exiting from a block. Third, since local variables do not have global side effects (such as assignment), the programmer can control the local logical complexity of the data objects referenced.

<u>Block Structure</u>	<u>Without Block Structure</u>
real a, b, x, y, z;	real a, b, x, y, z, temp;
,	
if a ≥ b then	temp :=x;
begin	if a < b then go to L1;
real temp;	x:=y;
temp :=x;	y:=z;
x:=y;	z:=temp;
y:=z;	go to L2;
z:=temp;	L1 :x:=z;
end	z:=y;
else	y:= temp;
begin	L2:
real temp;	
temp :=z;	
z:=y;	
y:=x;	
x:=temp;	
end	

Fig. 1. A program with and without block structure.

In order to appreciate the importance of block structure, let us look at a trivial piece of code in Algol 60 in its blocked and unblocked versions (See Fig. 1). In the overall structure of Fig. 1, a selection between two computations is readily visible in the block-structured version, but is rather obscure at first glance in the unblocked version, which in addition incurs the creation of three additional identifiers.

BLOCK STRUCTURE

Using Block Structure in Programs. The material given in this section has been adapted from Wegner (1971).

DECLARATIONS. In most block-structured languages a program is also a block. A block in turn is a list of declarations called the "block head," followed by a list of statements called the "block body" enclosed within the brackets **begin . . . end**. The rules of Algol 60 state that all identifiers (e.g., names of variables) used within a block have to be declared in its block head. Furthermore, *identifiers can be used only within the block in which they have been declared*. This block is their "scope" and they are said to be "local" to it.

In addition to scalar variables of types, **boolean**, **integer**, and **real**, arrays of each of these data types and procedures may also be declared in the block head. Each time a block is activated [discussed below], the expressions for the array bounds are newly computed, thereby allowing the aforementioned variable array bounds. Procedures may be of the statement type-returning no value (i.e., an extension of the Fortran subroutine concept) they may be of the function type, in which case they return a single value to their point of invocation. Both types of procedures may assign new values to

any number of nonlocal variables. They are then said to have a side effect. Function type procedures have a type associated with the values they return.

SCOPE RULES AND INITIALIZATION. As mentioned above, a block body is a sequence of statements and a block is a particular kind of statement. Therefore, blocks may be nested to any depth. This design has several consequences :

1. Although an identifier may be used only to name one object in a block, the same identifier may be used to name different objects in different blocks.

2. At run time, whenever a block is entered a new copy of all the variables declared in its block head is created. This set of objects (collectively known as the activation record of the block) defines a particular instance of the block in time. The activation records comply with a strict stack discipline of first-in-first-out (see next section). Therefore, if an identifier is declared in an outer block as well as in the currently active block, only the currently declared identifier may be referenced.

3. Labels in front of blocks may be implicitly declared (as in our example below). Since blocks may be entered only from their block head, a major design decision is how to treat **goto**'s. Two basic

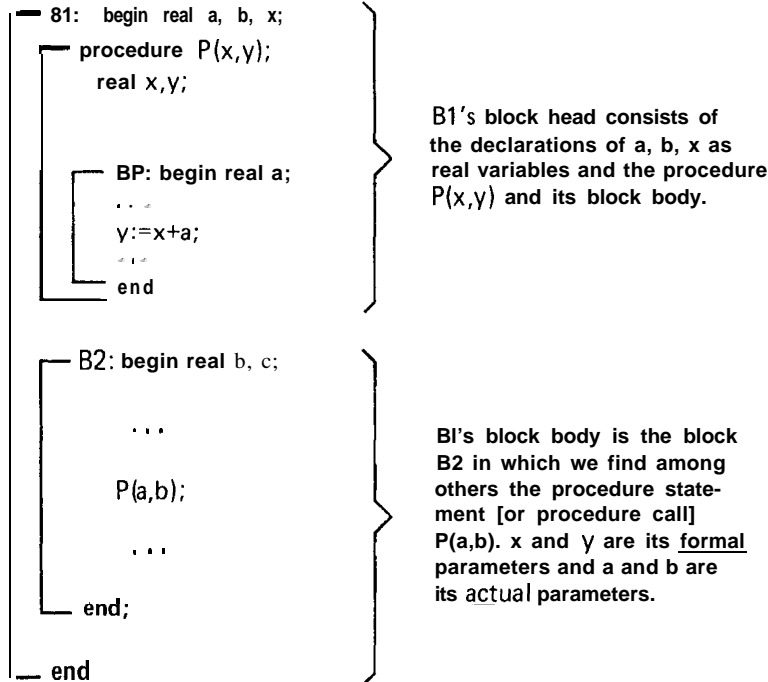


Fig. 2. Procedures in block structure.

strategies may be employed: (1) The conservative strategy is not to allow `goto`'s to the middle of the block, thereby maintaining the stacklike discipline of Algol 60; (2) the liberal strategy is to assume that all information pertaining to the state of a computation at a particular statement in the program is carried in the label of that statement. This results in a rather interesting but error-prone language design.

Identifiers of type procedure are said to be "initialized" to the value of their body. They cannot be redefined at run time. All other identifiers are "uninitialized" and must be explicitly assigned a value via the execution of an assignment statement.

To summarize, let us look at the example in Fig. 2. Note that the identifier b is declared once in the B1 block and again in the B2 block. Thus, $P(a,b)$ refers to the local declaration of b which is created upon entering the B2 block. Also, a in BP has scope BP and is a local variable used in computing $P(a,b)$.

Theoretical Models of Block Structure Execution. The descriptions and examples in this section are from Wegner (1971).

Several models have been suggested for describing the run-time execution of programs written in block-structured languages. We will look at two of them: the stack model proposed by Dijkstra (1967) and the contour model developed by Johnson (1971).

The *stack model*, $SM = (P,C,S)$, consists of

- P: a program component, which remains unchanged throughout the execution of the program.
- C: a control component, consisting of two pointers: an instruction pointer ip , pointing to the current instruction, and an environment pointer ep , pointing to the current environment.

S: a stack of activation records containing all the data the program operates on.

In terms of the preceding example, we have the following in Fig. 3: At the time instant modeled, execution is at statement $b := x + a$, within block BP. Previously, blocks P, B2, and B1 have been properly entered.

Using our model, we can follow the execution of block B1 and its contained blocks by setting the $ip = 1$ and $ep = \text{null}$ and taking a sequence of snapshots of our model as the computation proceeds in time. The snapshots should be taken after any of the following actions has taken place: block entry or block exit; procedure call or procedure return.

The model works as follows: When ip points to statement BP, which is a block, then an activation record is created (consisting of sufficient memory locations for storing the values of all the identifiers declared in BP) and placed on top of the stack S. Next, the ip is pointed to the first statement in B1 and the ep is pointed to the first location of the newly created activation record. Upon exiting from BP, the current activation record is deleted from S; ep is pointed to the next lower activation in S; and ip is pointed to the next statement in sequence. Analogous processes occur upon procedure call and procedure return.

Since the stack and contour models are similar in many ways, only those features unique to the contour model are described in the following discussion. The contour model, $CM = (P,C,D)$, consists of

- P: the program component, represented by a two-dimensional picture in which the nesting of blocks and procedures is denoted by the contours enclosing the statements of the block.

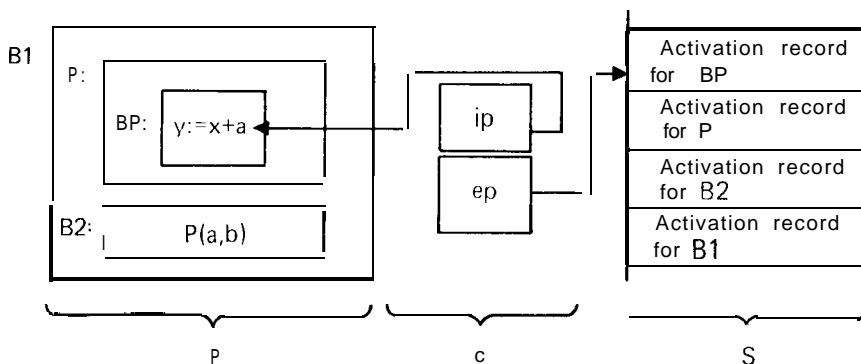


Fig. 3. Stack model of block structure.

BLOCK STRUCTURE

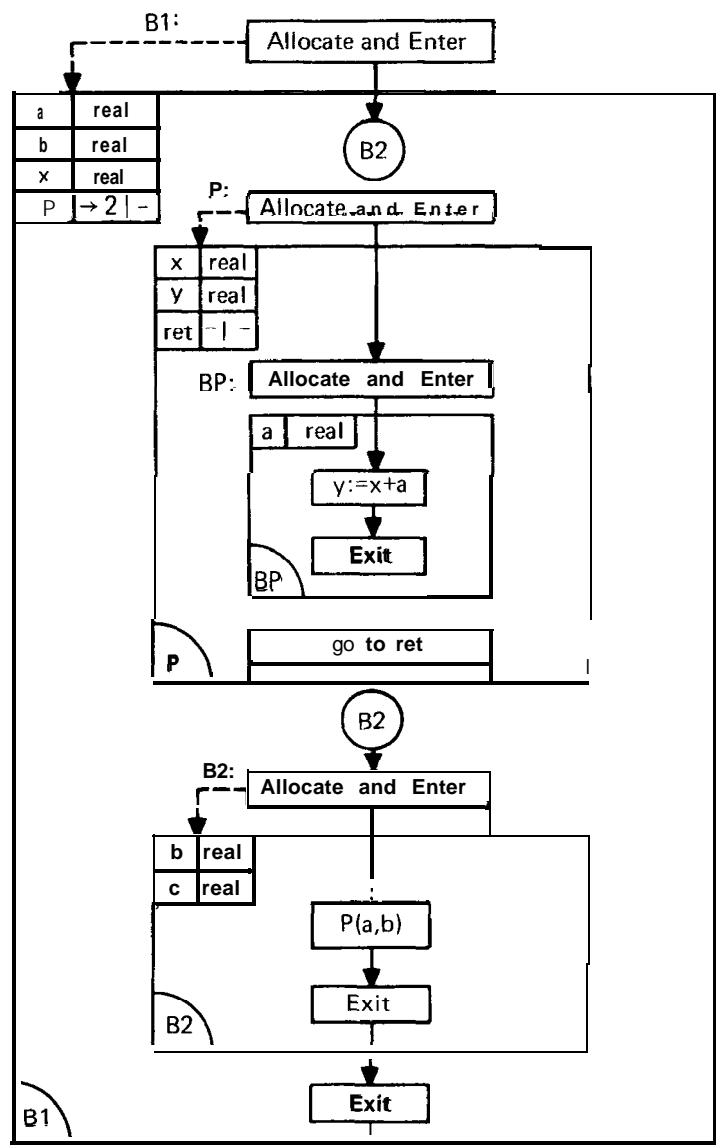


Fig. 4. Contour model of block structure.

- D: the data component, represented by a nesting of activation records enclosed in their respective contours.
- C: the control component, represented by the symbol π (for processor) drawn inside the contour of the current activation record.

The symbol π points to the value of the current **ip**, but the **ep** is implicit because it is bound by the innermost contour.

The contour model of our sample program is given in Fig. 4. Instead of explicit block entry and exit, the contour model employs the instructions *allocate* and *enter* and exit, respectively. To enter a block or a procedure, the *allocate* and *enter* employs an *identifier definition template* of the block to create the proper activation record. An *exit* instruction is inserted at the end of each block. Likewise, each procedure has a label-valued variable, represented by the symbol "ret," which is initialized at entry time to a return label. (This procedure is analogous to the one employed by most current machine languages to implement the subroutine feature.)

Snapshots of execution states are represented by contour diagrams. Upon entering block B1 the diagram is as shown in Fig. 5. The cells *a, b, x* are uninitialized, while the cell *P* is initialized to **ip** = 2, the location where *P(x,y)* is defined and the **ep** arrow points to the contour in which *P(x,y)* is declared.

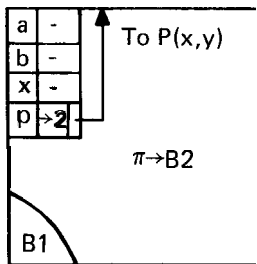


Fig. 5. Contour diagram snapshot of execution state.

The major advantage of the contour model is that the two-dimensional contour diagrams used to represent snapshots of the execution show explicitly the run-time identifier-accessing relation. They do not, however, give a direct representation of the order in which activation records are entered and exited. That is the strength of the stack model.

REFERENCES

1964. Randell, B., and L. S. Russell. *ALGOL 60*

Implementation. New York: Academic Press.

1967. Dijkstra, E. W. "Recursive Programming." In S. Rosen (Ed.) *Programming Systems and Languages*. New York: McGraw-Hill.

1967. Ekman, T., and C. E. Froberg. *Introduction to ALGOL 60*. London: Oxford University Press. (This book contains the complete text of the revised ALGOL 60 report by Naur et al., as well as a beautiful discussion of the control and data structures of ALGOL 60.)

1968. Wegner, P. *Programming Languages, Information Structures and Machine Organization*. New York: McGraw-Hill.

1971.—. "Structured Model Building in Computer Science." Providence, R. I.: Brown University, Dept. of Applied Mathematics.

1971. Johnson, J. B. "The Contour Model of Block Structured Processes," in J. T. Tou and P. Wegner (Eds.), *Proceedings of a Symposium in Programming Languages*. Gainesville: University of Florida, pp. 55–82.

A. N. GILEADI

BNF. See BACKUS-NAUR FORM.

BOOLE, GEORGE

For article on related subject see **BOOLEAN ALGEBRA**.

George Boole (b. Lincoln, England, 18 15; d. Cork, Ireland, 1864) was one of those rarities in an era of increasing specialization: the self-taught man who followed his own path to the penetration of territory untouched by his contemporaries. Due to the family's sparse financial resource, Boole's formal education was limited to elementary school and a short stint in a commercial school. Beyond this he was almost totally self-educated.

Boole's first scientific publication was an address on Newton to mark the presentation of a bust of Newton to the Mechanics Institution in Lincoln. In 1840 he wrote his first paper for the *Cambridge Mathematical Journal*. In 1849, despite his lack of formal training, he was appointed to a professorship of mathematics in the newly established Queen's College, Cork, Ireland,

BOOLEAN ALGEBRA

During his career he published approximately fifty scientific papers, two textbooks (on differential equations, 1859; and finite differences, 1860), and his two famous volumes on mathematical logic (see references). In 1844 the Royal Society awarded him a medal for his papers on differential operators, and in 1857 they elected him a Fellow. He was married in 1855 to Mary Everest, a niece of Sir George Everest after whom Mount Everest was named.

Although Boole made significant contributions in a number of areas of mathematics, his immortality stems from his two works that gave decisive impetus to the need to express logical concepts in mathematical form: "The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning" (1847) and "An Investigation of the Laws of Thought, on Which are Founded the Mathematical Theories of Logic and Probability" (1854). Through these works he truly became the founder of modern symbolic logic. He reduced logic to a propositional calculus, now called boolean algebra, which was extremely simple and perhaps too strongly based upon classical logic.

Under the influence of his work, a school of symbolic logic evolved, which made a determined effort to unify logic and mathematics. As is usual, the impact of this effort was not realized until the latter part of the nineteenth century. Although de Morgan and Jevons expounded on his work during Boole's lifetime, it remained for Frege, Peano, and C.S. Peirce to relight the torch that finally led to the "Principia Mathematica" (1910–1913) of Russell and Whitehead.

Boole's discovery that the symbolism of algebra could be used in logic has had wide impact in the twentieth century. Today, boolean algebra is important not only in logic but also in the theory of probability, the theory of lattices, the geometry of sets, and information theory. It has also led to the design of electronic computers through the interpretation of boolean combinations of sets as switching circuits. For example, the logical sum of two sets corresponds to a circuit with two switches in parallel and the logical product corresponds to a pair of switches in series.

REFERENCE

1970. Broadbent, T. A. A., "George Boole." In *Dictionary of Scientific Biography*, vol. II, pp. 293–298. New York: Scribners. (This is an outstanding biography with an excellent bib-

liography of both primary and secondary sources.)

H. TROPP

BOOLEAN ALGEBRA

For articles on related subjects see **ARITHMETIC**, **COMPUTER**; and **LOGIC DESIGN**.

For related biographical information see **BOOLE, GEORGE**.

The concept of a boolean algebra was first proposed by the English mathematician George Boole in 1847. Since that time, Boole's original conception has been extensively developed and refined by algebraists and logicians. The relationships among boolean algebra, set algebra, logic, and binary arithmetic have given boolean algebras a central role in the development of electronic digital computers.

Set Algebras. The most intuitive development of boolean algebras arises from the concept of a set algebra. Let $S = \{a, b, c\}$ and $T = \{a, b, c, d, e\}$ be two sets consisting of three and five elements, respectively. We say that S is a "subset" of T , since every element of S (namely, a , b , and c) belongs to T . Since T has five elements, there are 2^5 subsets of T , for we may choose any individual element to be included or omitted from a subset. Note that these 32 subsets include T itself and the empty set, which contains no elements at all. If T contains all elements of concern, it is called the "universal set." Given a subset of T , such as S , we may define the "complement" of S with respect to a universal set T to consist of precisely those elements of T which are not included in the given subset. Thus, S as above defined has as its complement (with respect to T) $\bar{S} = \{d, e\}$. The "union" of any two sets (subsets of a given set) consists of those elements that are in one or the other or in both given sets; the "intersection" of two sets consists of those elements that are in both given sets. We use the symbol \cup to denote the union, and \cap to denote the intersection of two sets. For example, if $B = \{b, d, e\}$, then $B \cup S = \{a, b, c, d, e\}$, and $B \cap S = \{b\}$.

While other set operations may be defined, the operations of complementation, union, and intersection are of primary interest to us. A boolean algebra is a finite or infinite set of elements together

with three operations—negation, addition, and multiplication—that correspond to the set operations of complementation, union, and intersection, respectively. Among the elements of a boolean algebra are two distinguished elements: 0, corresponding to the empty set; and 1, corresponding to the universal set. For any given element a of a boolean algebra, there is a unique complement a' with the property that $a + a' = 1$ and $aa' = 0$. Boolean addition and multiplication are associative and commutative, as are ordinary addition and multiplication, but otherwise have somewhat different properties. The principal properties are given in Table 1, where a , b , and c are any elements of a boolean algebra,

Table 1

Distributivity :	$a(b + c) = ab + ac$ $a + (bc) = (a + b)(a + c)$
Idempotency:	$a + a = a$ $aa = a$
Absorption laws :	$a + ab = a$ $a(a + b) = a$
DeMorgan's laws:	$(a + b)' = a'b'$ $(ab) = a' + b'$

Since a finite set of n elements has exactly 2^n subsets, and it can be shown that the finite boolean algebras are precisely the finite set algebras, each finite boolean algebra consists of exactly 2^n elements for some integer n . For example, the set algebra for the set T defined above corresponds to a boolean algebra of 32 elements. Tables 2 and 3 define the boolean operations for boolean algebras of two and four elements, respectively.

Table 2. Two elements.

$a + b$	0 1	$a \cdot b$	0 1	a	a'
0	0 1	0	0 0	0	1
1	1 1	1	0 1	1	0

Table 3. Four elements.

$a + b$	0 p p' 1	$a \cdot b$	0 p p' 1	a	a'
0	0 p p' 1	0	0 0 0 0	0	1
p	p p 1 1	p	0 p 0 p	p	p'
p'	p' 1 p' 1	p'	0 0 p' p'	p'	p
1	1 1 1 1	1	0 p p' 1	1	0

While it is possible to use a different symbol to denote each element of a boolean algebra, it is often more useful to represent the 2^n elements of a finite

boolean algebra by binary vectors having n components. With such a representation the operations of the boolean algebra are accomplished componentwise by considering each component as an independent two-element boolean algebra. This corresponds to representing subsets of a finite set by binary vectors. For example, since the set T has five elements, we may represent its subsets by five-component binary vectors, each component denoting an element of the set T . A numeral 1 in the i th component of the vector denotes the inclusion of the i th element of that particular subset; a 0 denotes its exclusion. Thus, the subset $S = \{a, b, c\}$ has the binary vector representation $\{1, 1, 1, 0, 0\}$. The set operations become boolean operations on the components of the vectors. This representation of sets, and the correspondence to boolean or logical operations, is very useful in information retrieval. Because of it, sets of document and query characteristics may be easily and rapidly matched.

Elementary Logic. In information retrieval work, and in identifying boolean algebras as set algebras, we find that various logical connectives, such as “and,” “or,” and “not,” recur frequently. Thus, it is not surprising to find that the two-element boolean algebra can be identified with elementary logic or propositional calculus. A “proposition” is a statement that can be said to be either true or false. We will denote propositions by letters such as p , q , and r .

The connectives or operators “and” and “or” combine two such propositions into a new one. If we consider two propositions, p and q , each may, independently of the other, assume the value true (T) or false (F). Hence, together the ordered pair $\langle p, q \rangle$ may assume $2 \cdot 2 = 4$ combinations of truth values: $\langle T, T \rangle$, $\langle T, F \rangle$, $\langle F, T \rangle$, and $\langle F, F \rangle$. If \circ denotes a binary operator, then $p \circ q$ may assume either (T) or (F) independently for each of these four T-F combinations. Thus we can define $2^4 = 16$ distinct binary logical operators, as shown in Table 4. Of the 16 binary logical operators that can be defined, 5 are commonly used and are more than sufficient to define the remaining operators.

Table 4

$p \ q$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T T	T	T	T	T	T	T	T	T	F	F	F	F	F	F	F	F
T F	T	T	T	T	F	F	F	F	T	T	T	T	F	F	F	F
F T	T	T	F	F	T	T	F	F	T	T	F	F	T	T	F	F
F F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F

BOOLEAN ALGEBRA

The "negation" or "not" operation, $\sim p$, is defined to form a proposition that is true precisely when the proposition p is false, and false whenever p is true. If we equate the truth values "true" and "false" with the boolean values 1 and 0, respectively, then we find that negation corresponds to boolean complementation. That is, $\sim p$ replaces the value "true" with "false," and vice versa, just as p replaces the value "1" with "0," and vice versa. (In Table 4, column 13 is $\sim p$.)

The logical "conjunction" or "and," $p \wedge q$, forms a proposition that is true precisely when both p and q are true, and false otherwise. This corresponds to the boolean operation of multiplication, with the boolean expression pq having the value 1 if and only if both p and q have as value the 1. (See Table 4, column 8.)

In ordinary usage the word "or" has two distinct meanings, referred to as the "inclusive or" and the "exclusive or." In the inclusive sense, the statement " p or q " is true if p or q or both are true; in the exclusive sense, the same statement is true if either p or q , but not both, are true. The logical "disjunction" or "or," $p \vee q$, is defined to be the inclusive "or." That is, $p \vee q$ is true precisely when at least one of the statements p and q is true. Thus, this operation corresponds to boolean addition as we have defined it. (See Table 4, column 2.)

The exclusive "or," $p \not\equiv q$, is commonly called "inequivalence," since it defines a proposition that is true precisely when p and q have opposite or inequivalent truth values. This corresponds to any of several more complex boolean operations such as $pq' + p'q$, and $(p + q)(pq)'$. (See Table 4, column 10.)

The remaining conventional logical operator is the "conditional" or "implication," $p \supset q$, corresponding to the statement "if p then q ." The conditional proposition $p \supset q$ takes the value "false" if p is true and q is false, and takes the value "true" otherwise. Thus, it corresponds to the boolean op-

eration $p' + q$. Note that if p is false, then $p \supset q$ is true, regardless of the value of q . This corresponds to the statement that one can prove anything (q , whether true or false) from a false hypothesis (p). (See Table 4, column 5.)

While the logical operators that we have defined suffice to define all logical operators, it is only necessary to use two of the above operators, namely, negation, and one of the operators conjunction, disjunction, or conditional. However, of importance to computer design is the fact that we can define all logical operators in terms of one basic operator, either the "nand" or the "nor" operator. These are the negation of the conjunction and disjunction operators, respectively. That is, the "nand" operator defines a statement, $p \mid q$, which has the value "false" precisely when both p and q are true, and the value "true" otherwise. The "nor" operator defines a statement, $p \downarrow q$, which has the value "true" precisely when both p and q are false, and the value "false" otherwise. (See Table 4, columns 9 and 15.)

Truth Tables. A truth table gives the truth values of a logical expression for each combination of the truth values of its variables. Thus, for a logical expression in n variables, the truth table contains 2^n lines, one for each combination of truth values of its variables. Since the truth value of an expression is determined from the truth values of various subexpressions, the truth table may be given in an extended form, which explicitly lists all subexpressions, a standard form in which the subexpressions are not separately listed and a condensed form in which the lines of the table are compressed by indicating the truth value of certain critical subexpressions. Tables 5 through 7 illustrate these three forms of truth table for the logical expression $(p \equiv q) \supset \sim(((p \vee \sim r) \wedge (\sim p \vee q)) \supset r)$.

In each of these three tables the truth values for the given expression are in the boxed column. In the

Table 5

$p \quad q \quad r$	(1) $p \equiv q$	(2) $p \vee \sim r$	(3) $\sim p \vee q$	(4) $(2) \wedge (3)$	(5) $(4) \supset r$	(6) $\sim(5)$	Expression (1) \supset (6)
T T T	T	T	T	T	T	F	F
T T F	T	T	T	T	F	T	T
T F T	F	T	F	F	T	F	T
T F F	F	T	F	F	T	F	T
F T T	F	F	T	F	T	F	T
F T F	F	T	T	T	F	T	T
F F T	T	F	T	F	T	F	F
F F F	T	T	T	T	F	T	T

Table 6

p	q	r	$(p \equiv q)$	\supset	\sim	$((p \vee \sim r) \wedge (\sim p \vee q)) \supset r$
T	T	T	T	F	F	T
T	T	F	T	T	TT	T
T	F	T	F	T	F	T
T	F	F	F	T	F	T
F	T	T	F	T	F	T
F	T	F	F	T	T	F
F	F	T	T	F	F	T
F	F	F	T	T	T	F

Table 7

p	q	r	$(p \equiv q)$	\supset	\sim	$((p \vee \sim r) \wedge (\sim p \vee q)) \supset r$
T	F	-	F	T	-	-
F	T	-	F	T	-	-
F	F	-	-	T	T	T
-	T	F	-	T	T	T
T	T	T	T	F	F	---
F	F	T	T	F	F	---

condensed form, Table 7, each line of the table may represent one or more lines of the uncondensed table. For example, the first line of Table 7 represents the two lines TFT and TFF of the uncondensed table. In this particular example, the line FTF is represented three times, namely, in lines 2, 3, and 4 of the Table 7. Also in the condensed table, the dashes represent values that are immaterial and hence do not need to be calculated. For example, in the first line of Table 7, since $p \equiv q$ is false, we know that the entire expression has the value "true," regardless of the value of the remaining portion of the expression.

The truth table for an unknown logical function can be used to generate an expression for that function. The expression thus generated is called a "disjunctive normal form" or, in boolean algebra, a "sum of products form." The development of this expression is illustrated in Table 8. For each line of the table wherein the unknown function has the value "true," an expression is formed by taking the conjunction of all variables that are true in that line and the negations of all variables that are false in that line. The expression for the function f is then the disjunction of all expressions formed for the single lines. In Table 8, f is given in this form, and in the corresponding boolean algebra form, as well as in a shorter form developed by direct inspection of the function values. (Equivalence, \equiv , is defined by column 7 of Table 4.)

The development of the disjunctive normal form shows that the logical operators conjunction,

Table 8

p	q	r	$f(p,q,r)$	Generated expression
T	T	T	F	
T	T	F	T	$p \wedge q \wedge \sim r$
T	F	T	T	$p \wedge \sim q \wedge r$
T	F	F	F	-
F	T	T	T	$\sim p \wedge q \wedge r$
F	T	F	F	-
F	F	T	F	-
F	F	F	T	$\sim p \wedge \sim q \wedge \sim r$

$$f(p,q,r) = (p \wedge q \wedge \sim r) \vee (p \wedge \sim q \wedge r) \vee (\sim p \wedge q \wedge r) \vee (\sim p \wedge \sim q \wedge \sim r)$$

$$f(p,q,r) = pqr' + pq'r + p'qr + p'q'r'$$

$$f(p,q,r) = p \equiv (q \neq r)$$

Table 9

	\wedge, \sim	\vee, \sim
$\sim p$	$\sim p$	$\sim p$
$p \wedge q$	$p \wedge q$	$\sim(\sim p \vee \sim q)$
$p \vee q$	$\sim(\sim p \wedge \sim q)$	$p \vee q$
$p \supset q$	$\sim(p \wedge \sim q)$	$\sim p \vee q$
$p \equiv q$	$\sim((p \wedge q) \wedge \sim(p \wedge \sim q))$	$\sim(p \vee q) \vee \sim(\sim p \vee \sim q)$

Table 10

	\downarrow	\downarrow
$\sim p$	$p \downarrow p$	$p \downarrow p$
$p \wedge q$	$(p \downarrow q) \downarrow (p \downarrow q)$	$(p \downarrow p) \downarrow (q \downarrow q)$
$p \vee q$	$(p \downarrow p) \downarrow (q \downarrow q)$	$(p \downarrow q) \downarrow (p \downarrow q)$

disjunction, and negation are sufficient to develop an expression for any logical function. Furthermore, we may use DeMorgan's laws to transform conjunctions to disjunctions, or vice versa. Thus, as we previously asserted, any logical function can be developed from the operators negation and either conjunction or disjunction. Table 9 shows the development of the five common logical operators in terms of these two minimal combinations of operators. In turn, Table 10 shows the development of negation, conjunction, and disjunction in terms of both the "nand" and the "nor" operators, thus indicating that every logical operator can be defined in terms of either one of these latter two operators.

Computer Arithmetic. The identification of the logical constants **T** and **F** with the boolean constants 1 and 0, respectively, leads to the development of the arithmetic properties of the computer in terms of its logical or boolean operators. In binary arithmetic, the multiplication of bits is exactly the

BOOLEAN ALGEBRA

same as boolean multiplication: The product of two bits is 1 if and only if both bits are 1. However, the addition of two bits is quite different from boolean addition. This is apparent, since in boolean arithmetic $1 + 1 = 1$, while in binary arithmetic $1 + 1 = 10$.

We also observe that in binary arithmetic the sum bit is 1 if and only if one, but not both, summands have the value 1, while the carry bit is 1 if and only if both summands have the value 1. Thus, we can compute the sum bit by using the logical inequivalence (exclusive or) operation, and the carry bit by using the logical conjunction, or boolean multiplication operation. Finally, we observe that, since the negative of an integer is normally represented in the computer by a complementary bit pattern (1's complement), arithmetic negation can be accomplished by logical negation, or boolean complementation, with slight modification if 2's-complement arithmetic is used.

Logical Design. Logical design of a computer is the development of computer circuitry to perform the desired functions for the particular machine. It is necessary that the circuitry be accurate and reliable, and desirable that it be relatively simple so that it is inexpensive and easy to maintain. While logical design must include consideration of timing problems and the various electromechanical attachments to the computer, the heart of the problem resides in the development of logical circuitry to perform the desired functions.

Of the various devices designed to systematize study of the logic, the Venn diagram or Karnaugh map is particularly simple and highly effective for functions of 2, 3, 4, or 5 variables. However, the use of this device becomes increasingly difficult as the number of variables increases beyond five. The

classical Venn diagram consists of a rectangle representing the universe, containing a circle or other simple closed curve for each variable represented. The interpretation is that within the circle the given variable has the value 1, while outside it has the value 0. These circles are arranged in such a way as to include all possible combination of 1's and 0's for the variables. The Venn diagram for a **3-variable** problem is given in Fig. 1, with the various regions labeled in Fig. 1(a) and certain regions shaded to represent the boolean function $pq + pr + p'r$ in Fig. 1(b). In this form the Venn diagram is relatively ineffective for logical analysis. The varying shapes of the regions cause some difficulty in visualizing possible combinations of these regions, particularly if four or more variables are involved.

The Karnaugh map is a practical modification of the Venn diagram, with each region of the diagram represented by a square within a larger rectangle. The Karnaugh maps for 2-, 3-, and 4-variable problems are given in Fig. 2. The region represented by each square is determined by the product of the letters on the edges of the rectangle. For example, the square marked A in the 4-variable rectangle represents the region $pq'rs$. To represent a boolean function, say $pq + pr + q'r$, on a Karnaugh map, first expand each term of the functions to include all variables present:

$$\begin{aligned} pq + pr + q'r &= pq \cdot 1 + p \cdot 1 \cdot r + 1 \cdot q'r \\ &= pq(r + r') + p(q + q')r + (p + p')q'r \\ &= pqr + pqr' + pqr + pq'r + pq'r + p'q'r \\ &= pqr + pqr' + pq'r + p'q'r. \end{aligned}$$

Then mark each square corresponding to a term in the expanded expression.

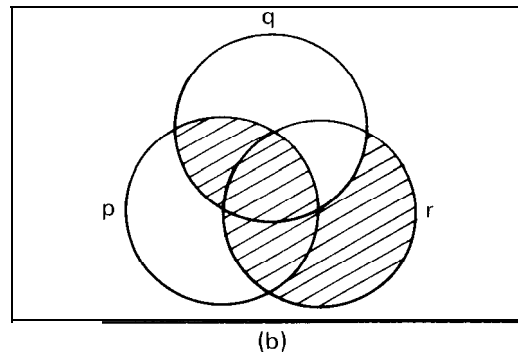
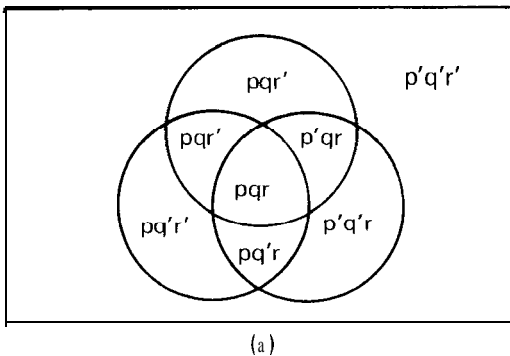


Fig. 1. Venn diagram.

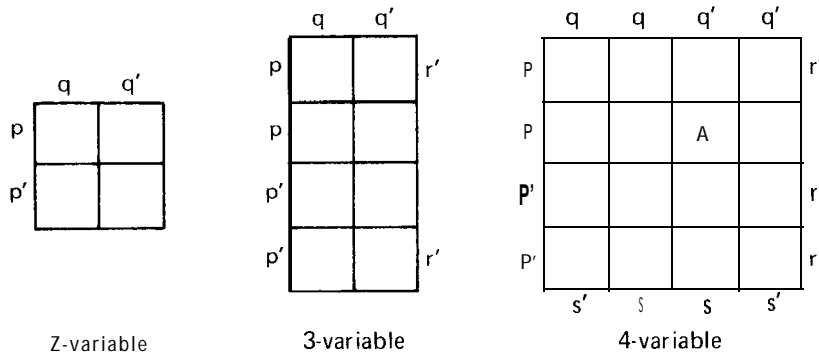


Fig. 2. Karnaugh maps.

Thus, the boolean function $pq + pr + q'r$ is represented by the squares marked "1" in Fig. 3, while 0's fill those squares not included in the representation. Note that $pq + q'r$ is also represented by the same four marked squares, and hence is equivalent to the given function. It is also possible to label a square d , denoting "don't care," if the value of that square is irrelevant to the particular function being represented.

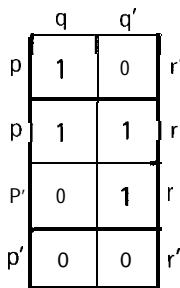


Fig. 3. Karnaugh map for pq plus pr plus $q'r$.

Minimization of Boolean Functions. In

the interest of economy it is often desirable to use the simplest possible expression for a boolean function in the design of computer circuitry. For example, since the expression $pq + pr + q'r$ is equivalent to the expression $pq + q'r$ in the sense that these expressions have the same value for given argument values, the former expression should be replaced by the latter whenever it occurs in a given circuit design. The determination of the simplest expression equivalent to a given one is known as "minimization." Minimization is understood to be with respect to a given function form, such as the sum of products form, since a change in permissible operators often permits one to find an expression that is simpler yet.

Karnaugh maps and a variety of algebraic or geometrical algorithms have been used to accomplish boolean function minimization.

REFERENCES

- 1965, McCluskey, E. J. *Introduction to the Theory of Switching Circuits*. New York: McGraw-Hill.
- 1966. Korfage, R. R. *Logic and Algorithms*. New York: Wiley.
- 1970. Mendelson, E. "Theory and Problems of Boolean Algebra and Switching Circuits." In *Schaum's Outline Theories*. New York: McGraw-Hill.
- 1972. Peatman, J. D. *The Design of Digital Systems*. New York: McGraw-Hill.

R. R. KORFHAGE

BOOTSTRAP

For article on related subject see **MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING**. For articles on related terms see **LANGUAGE PROCESSORS**; and **LOADER**.

Using some already running part of a language processor as a tool to get the rest of it running more easily (or using such a processor to get itself running on another machine without entirely rewriting it) is the programming counterpart of the apocryphal feat of lifting oneself by one's own bootstraps. These shortcuts are possible when the translator can be written using only a small but well-defined subset of

BREAKPOINT

the language it translates; when it has this property, only so much of it as is necessary to translate that subset need be hand coded, after which the description of the whole translator can be processed by the handwritten fragment. The output of this procedure is the whole translator in object form, "bootstrapped" into existence by the handwritten fragment of itself.

The steps involved in bootstrapping can be usefully represented by schematic diagrams in which a translator is shown as an oblong, within which an expression of the form 'A \rightarrow B' describes the translation it performs. The language in which the translator has been, or is to be, written is noted as a kind of subscript outside the oblong. For example, the representation in this notation of a **Fortran** compiler producing machine language (ML) code for computer X, and itself existing as an ML program for X, is

$$\boxed{\text{Fortran} \rightarrow \text{ML 'X'}}$$

ML 'X' (1)

Note that if the subscript is a machine language, the translator in question is running on a real machine and can be used immediately; if the subscript is a higher-level language, the translator is so far merely a source-language file that must itself be translated before it can be used to translate another source-language program. Given an immediately usable translator, the question of whether it can translate a given program is answered by matching the language in which the potential processee is written against part 'A' of the potential processor's 'A \rightarrow B' formula. If they are identical, or the former is a subset of the latter, the desired translation is feasible. In the notation (1) example, if the processee is a **Fortran** source program, it can be translated.

If we postulate that there exists some proper subset of the **Fortran** language in which a **Fortran** compiler can be written, the steps involved in bootstrapping into existence a **Fortran** compiler for and on machine X can be outlined in our notation. We must handwrite two programs:

$$\boxed{\text{FORTRAN} \rightarrow \text{ML 'X'}}$$

Fortran Subset

[This program written in the **Fortran** subset will translate any **Fortran** program to ML 'X'.]

(2)

$$\boxed{\text{FORTRAN subset} \rightarrow \text{ML 'X'}}$$

[This Program will translate any program ML 'X' written in the **Fortran** subset into ML 'X'.]

(3)

Then we translate (2) by means of (3), yielding

$$\boxed{\text{FORTRAN} \rightarrow \text{ML 'X'}}$$

ML 'X' (4)

which is the required product—a full Fortran-to-ML 'X' compiler running on machine X.

The question of when this approach is better than that of coding the desired product directly [as in notation (1)] is a complex one; some of the considerations involved are discussed in Halpern (1965).

"Boots trapping" is used also to describe the process whereby a programmed loader, whose job it is to load other pieces of software into a machine, gets itself in. This task, which at first glance seems to threaten infinite regression, is made possible by a miniloader built into the hardware. In a typical example, the computer will offer the operator the ability to load into core and execute some small number of instructions—six, say—simply by pushing a console button. These six "free" instructions would be used by the programmer to load and transfer control to a full programmed loader, which, when thus "bootstrapped" in, could load any desired program with such niceties as check sums, relocation, and external symbol linking.

REFERENCE

1965. Halpern, M. "Machine Independence: Its Technology and Economics," *CACM*, Vol. 8, No. 12 (December), pp. 782-785.

M. HALPERN

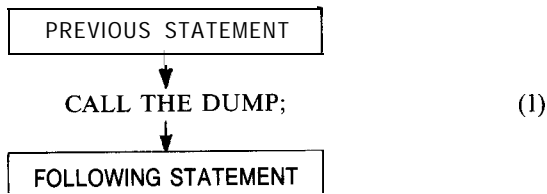
BREAKPOINT

For articles on related subjects see **COMPUTER**, **USING A**; **DEBUGGING**; and **DIAGNOSTICS**.

A breakpoint is a position in a program at which the programmer has arranged for normal

execution to be interrupted so that some type of external intervention can occur. This usually is associated with the debugging process in that the intervening activity is designed to provide status information and/or diagnostic data relative to the progress of the program up to that point. For example, the programmer may select one or more strategic places in the program where he would like to see a dump (i.e., a copy of the contents) of pertinent storage locations to assess the correctness of intermediate results.

In some systems the action at a breakpoint is performed automatically by a software component (an instruction inserted in the program for this purpose). For example, the programmer writing in PL/I for the IBM 360/370 series has direct access to a dump routine, with normal processing resuming after its completion. Accordingly, the breakpoint is set up like any other subroutine call:



In other types of breakpoints, the external action must be performed by an operator. Under these circumstances the action is independent of the user's program so that it is necessary for the operator to reactivate the program manually. An example of such a facility is seen in some dialects of Fortran, where one may write

PAUSE message (2)

with the result that the program will halt and the message associated with the particular **PAUSE statement** will be displayed. Prior to the resumption of processing (which is done manually), the operator may interject the appropriate action. Whatever their form, the statements that create the breakpoint are retained until the programmer has identified and corrected the difficulties. Conversely, breakpoints may be created anew when unanticipated troubles develop in a seemingly operational program.

Additional types of breakpoints may be set up in conjunction with hardware. These facilities generally take the form of bistable switches, which may be set externally and tested by special statements within the program. Thus, the position of one of the switches may determine whether or not the program

will halt at some point, to be restarted manually after some action is taken by the operator. Accordingly, that switch may be "on" during a debugging run; for normal execution, the switch is left in the "off" position, in which case the program will execute without interruption.

S. V. POLLACK

BRITISH COMPUTER SOCIETY (BCS)

For article on related subject see **INTERNATIONAL FEDERATION OF INFORMATION PROCESSING.**

The British Computer Society (BCS) was formed in September 1957 with the following main objectives;

1. To further the development and use of computational machinery, and the techniques related thereto.
2. To facilitate the exchange of information and views, and to inform public opinion on the subject.
3. To hold conferences and meetings for the reading of papers and delivery of lectures.
4. To publish information for the benefit of members.
5. To organize and conduct examinations, for members and others, in subjects requiring a knowledge of or otherwise in any way concerning the development and use of computational machinery and the techniques related thereto, and in any allied subjects.

A number of interested people, who foresaw the vital importance of computers to the community, met during the early 1950's to initiate lectures on different aspects of computing science and to discuss the problems of its application to industrial and commercial work. These conferees were, on the one hand, people with scientific and engineering interests, and on the other, members of the London Computer Group, which represented industry and commerce. As a result of these meetings, the British Computer Society, a company limited by guarantee, was formed on Oct. 14, 1957.

At a special meeting in May 1968, the Society decided to become a fully professional body. To this end, the Society has introduced examinations (held

BRITISH COMPUTER SOCIETY (BCS)

annually in April each year since 1969); adopted a Code of Conduct (February 1971); and produced a Code of Good Practice (January 1973).

The following have held the office of BCS president :

Professor **M.V. Wilkes**, 1957-1960
Sir Frank Yates, 1960-1961
D.W. Hooper, Esq., 1961-1962
R.L. Michaelson, Esq., 1962-1963
Sir Edward Playfair, 1963-1965
Sir Maurice Banks, 1965-1966
Earl of Mountbatten of Burma, 1966-1967
Dr. S. Gill, M.A., Ph.D., 1967-1968
B.Z. de Ferranti, M.A., **C.Eng.**, 1968-1969
The Earl of Halsbury, 1969-1970
A. d'Agapeyeff, Esq., 1970-1971
Professor A.S. Douglas, 1971-1972
G.J. **Morris**, Esq., 1972-1973
R.A. Barrington, Esq., 1973-1974
E.L. Willey, Esq., 1974-1975
C.P. Marks, Esq., 1975-

Organizational Structure. The British Computer Society is run by a Council, which consists of 47 members: 18 elected; 12 from the branches of the BCS; and 17 others, including officers, students, and specialist group representatives. The Council, which meets quarterly, operates through boards and committees. The main boards are the Membership Board, the Technical Board, the Education Board, and Branch Board.

Membership. The membership and education boards work in close collaboration to set the standards of experience and education required for Membership in the British Computer Society. The membership structure of the Society allows seven classifications, as follows:

FELLOW, Fellowship is by election from the Member grade. The minimum requirements are that Fellows must be over 30 and have eight years' experience in computing, five in a responsible position.

MEMBER, Applicants must be over 25, have passed (or have been exempted from) BCS Parts I and II, and have five years' experience in computing, or have seven years' experience.

LICENTIATE. Licentiates must be over 21, have passed (or been exempted from) BCS Parts I and II, but not both, and have three years' experience in computing.

ASSOCIATE, This is a holding grade by people who have passed Part I and are taking Part II. They

remain as Associate until they have the required three years experience for Licentiate grade or five years for Member grade.

AFFILIATE, This grade is for those who do not wish to become fully professional members of the BCS. This is also a grade for those with less than seven years' experience who will shortly be applying for transfer to higher grades, without having to take examinations.

STUDENT, A student must be over 17 years of age. He remains a student until he passes Part I of the BCS examinations, when he is eligible to become an Associate.

INSTITUTIONAL AFFILIATE. This grade accommodates corporate bodies, companies, educational institutions, societies, etc.

Education. The responsibility for the Society's educational activities lies with the Education Board and its committees, assisted by a full-time education department. Education liaison officers appointed by each branch play a valuable role in **communication** between the Society and educational establishments throughout the country. Information is also provided on career prospects in the computer field, including presentations to schools and colleges.

The Society has played, and is sustaining, an important part in encouraging the spread of computer knowledge through its Schools Committee, which consists of people from education administration, from the teaching profession and industry, and the Group for Computer Education, also affiliated with the Society. The Group, with a membership of **2,700**—comprising secondary school teachers, college lecturers, and training officers from industry—has an international reputation through the publication of its quarterly bulletin, *Computer Education*; almost a quarter of its membership is drawn from abroad.

The Society plays a major role in setting and maintaining standards, at many levels of competence, by its representation on the advisory committees of national examining and educational **bodies**.

The work of the Society's members, individually or as government representatives, in international organizations concerned with education, places it at the international center of computer education circles.

The Society's annual publication, *The Educational Yearbook*, is the definitive work of reference for computer education and is international in its coverage.

The Society Examination. The Society's examination, set in two parts, is designed to assess the candidate's understanding of the underlying principles of the discipline, his ability to reason and to evaluate information, and his capacity for application of his knowledge to the solution of both practical and theoretical problems.

The Part I examination, set at the level of the Higher National Diploma, requires candidates to take two compulsory papers covering the general knowledge that all computer professionals should have, together with two papers from a number of widely defined areas of more specialized computer knowledge (computer technology, programming, data processing, analysis and design of systems, computational methods, and analog and hybrid computing). The Part II examination, set at the level of a university honors degree, requires candidates to take two papers in one area and one paper in a second, more specialized, area than those defined for Part I (digital computer technology, systems programming, data processing and information systems, advanced programming theory, data processing management, numerical analysis, and hybrid computing).

Branch Activities. There are 35 branches in the United Kingdom, one in Hong Kong, and one in Zambia. The branches, staffed entirely by volunteers, arrange programs of lectures and visits to installations.

Technical Activities. The Society is actively engaged in formulating and expressing professional viewpoints on a variety of subjects of importance. This is a primary responsibility of the Technical Division and its specialist committees.

In addition to its technical work in the United Kingdom, the Technical Board coordinates the work of its representatives on the IFIP technical committees and working groups and on the International Standards Organization. The Technical Board also coordinates the work of its 37 specialist groups, which study aspects of computer science ranging from advanced programming to urban planning.

Publications. In addition to the publications mentioned above, the Society has three other major publications. The Computer *Journal* is published quarterly. It contains articles and papers on scientific, business, and commercial subjects related to computers, together with reviews of the most important books and other publications in the field. The *Computer Bulletin* is also published quarterly and

contains articles of a more general, tutorial nature than those in the *Journal*. The weekly *Computing* is published for the Society by Haymarket Publishing Group and contains news of the European computer industry and articles of a general nature on computer applications. It is the main communications link between the Council and the members of the Society, and carries reports of branches and specialist groups as well as advance notice of all the Society's activities.

Along with these publications are the reports of the proceedings of the Society's many specialized conferences, together with authoritative handbooks such as *Code of Good Practice, Privacy and the Computer-Steps to Practicality*, and *Character Recognition*, all published by the Society.

Conferences. The Society, in conjunction with its partners, presents a number of conferences of interest to the computing fraternity and conducts techniques Workshops on a number of specialized subjects. It also provides a number of highly specialized conferences on such subjects as Codasyl and Relational Data Base concepts.

I. L. AUERBACH

BUFFER

For article on related subject see **INPUT-OUTPUT CONTROL SYSTEM**.

For article on related term see **FIFO-LTFO**.

A buffer is an area of storage which temporarily holds data that will be subsequently delivered to a processor or input/output (I/O) transducer. Buffers exist as an integral part of many transducers; e.g., bits arriving serially over a telephone line are collected in a buffer before the appropriate teleprinter character is activated. Similarly, the bits representing a given keyboard stroke remain in a buffer while being serialized for transmission. Since the buffer is an integral part of the transducer, it is usually dedicated to the transducer and not shared with any other device.

Buffers are also used in conjunction with the input/output control system (IOCS) to hold the data which is the object of various I/O commands. In this case, the buffer is usually a portion of main storage and is often dynamically allocated and freed by software. In either case, a buffer exists in order to

BUFFER

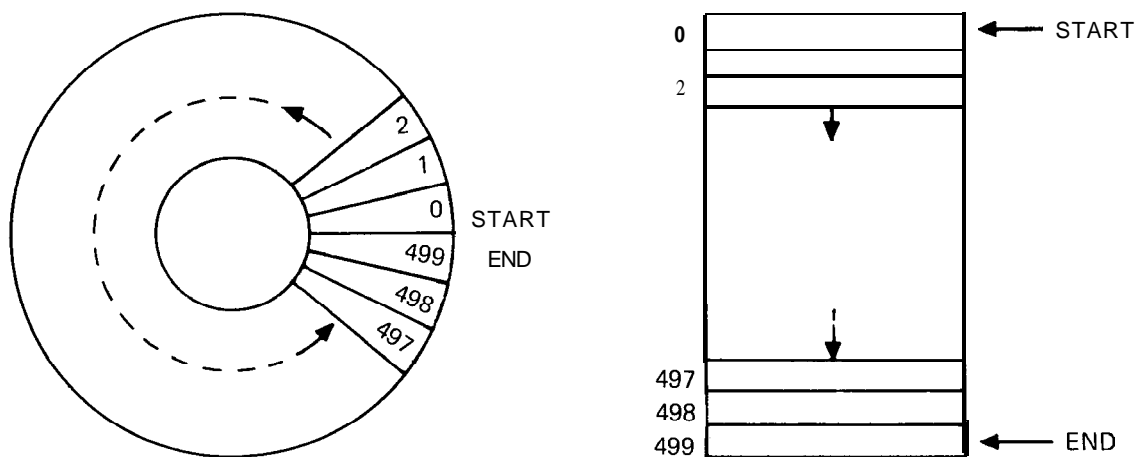


Fig. 1. Circular buffer organization shown logically (left) and as it actually appears in memory (right).

accommodate the different rates at which data is produced or consumed by the processor or transducers involved.

In a typical situation, a processor will be capable of producing data three orders of magnitude faster than a transducer can accept it. In order to make most efficient use of the processor, the data will be placed in a buffer and its location made known to the transducer. The transducer then proceeds to empty the buffer while the processor is freed for other work.

Various buffering techniques have evolved in IOCS. These techniques can be analyzed according to the policy used for (1) receiving data from the producer and (2) delivering data to the consumer.

When receiving data, two techniques are common: (1) a pool of buffers and (2) circular buffering. With the buffer-pooling technique a number of buffers are available to the IOCS. Usually, each buffer is large enough to hold the single physical record that is being transferred. When a record is produced, a buffer is taken from the pool and used to hold the data. Data is then consumed on a first-in, first-out basis, and when all data in a buffer has been transmitted, the buffer is returned to the pool.

Circular buffering, in contrast, typically uses a single buffer, usually larger than a single physical record. The basic strategy is to give the appearance that the buffer is organized in a circle, with data "wrapping around" as shown in Fig. 1. This appearance of circular organization is accomplished by using two pointers, **IN** and **OUT**, associated with the buffer; the starting and ending addresses of the

buffer (**START** and **END**) are also known. Initially, **START = IN = OUT**. Data received from the producer fills the buffer, starting from **START** and incrementing the pointer **IN**. The consumer takes data from the buffer, incrementing the pointer **OUT** (and taking care not to go past **IN-1**). When the last word of the buffer has been filled (**IN = END**), then **IN** is reset to **START** and subsequent data will wrap around to the start of the buffer.

Similarly, when **OUT** reaches **END**, it is reset to **START** and also wraps around. Clearly, the following restrictions hold:

1. If **IN > OUT**, then **OUT** must not become greater than **IN-1**.
2. If **OUT > IN**, then **IN** must not become greater than **OUT--**.

If either of these two conditions is violated, then the consumer is trying to access data that has not been produced, *or* the producer is attempting to store over data that has not yet been consumed.

Data is delivered to the consumer either by moving it to a storage area provided by the consumer or by providing the consumer with a pointer to the data in the buffer. In the latter case, the consumer will frequently provide the IOCS with additional space, which becomes the new buffer. Such a technique is often called "exchange buffering."

R. W. TAYLOR

BUG

For articles on related subjects see **DE-BUGGING**; and **GLITCH**.

A "bug" is an error in either the mechanics or the logic of a computer program. The term arose during World War II, in connection with electronic testing, as an outgrowth of "debug" which was a synonym for "troubleshoot". The earliest computer programmers, who were frequently the designers and builders of the computers, transferred the term to its present usage.

Most mechanical bugs can be detected during the translation from the symbolic languages that programmers use into the (binary) language which is eventually executed. For example, the proper symbolic code for addition on many machines is **ADA** ("add to accumulator"). If the programmer mistakenly writes **ADD**, this bug will be detected, an error message will be printed, and execution of the program will be halted, since the attempted operation code is illegal.

A bug is also created, and a more serious one, if the programmer writes the legal code **SBA** ("subtract

from accumulator") when he meant to write **ADA**. This is a logical bug, and no coding system can catch such an error.

Properly speaking, the elimination of the first type of bug is the process of debugging, whereas the detection and elimination of the second type is the process of program testing. Program bugs can be so extremely subtle that they may resist great efforts to eliminate them. It is commonly accepted that all very large computer programs (such as compilers) have bugs remaining in them. The number of possible paths through a large computer program is enormous, and it is physically impossible to explore all of them. The single path containing a bug may not be followed in actual production runs for a long time (if ever) after the program has been certified as correct by its author or others.

F. GRUENBERGER

BUSINESS DATA PROCESSING. See

ADMINISTRATIVE-BUSINESS APPLICATIONS.

CAD . See COMPUTER-AIDED DESIGN.

CAL See COMPUTER-ASSISTED INSTRUCTION.

CM I. See COMPUTER-MANAGED INSTRUCTION.

CPU. See CENTRAL PROCESSING UNIT.

CPM. See PERT/CPM.

CACHE MEMORY

For articles on related subjects see ASSO-CIATIVE MEMORY; MEMORY: Main; and STORAGE HIERARCHY.

Cache memory is a mechanism interposed in the memory hierarchy between main memory and the CPU to improve effective memory transfer rates and accordingly raise processor speeds. The name refers

to the fact that the mechanism is essentially hidden and appears transparent to the user, who is aware only of an apparently higher speed large main memory. The cache memory is implemented in high-speed semiconductor technology, often with an associative address selection, whereas the main memory may be either core or semiconductor.

The cache concept anticipates the likely reuse of an organized temporary copy of data in main storage which has been recently used by the CPU. The concept is further extended to include data that is adjacent to data that has been used. Accordingly, it is usual to transfer several words from main store to the cache even though the immediate need is for only one word. If the required word is included in a stream of sequential instructions, it is likely that subsequent words will also be used; if these are retrieved with the required first word, repeated accesses to main memory will be unnecessary.

When used in conjunction with a cache store, the main memory is equipped to provide several words in address sequence when one of them is required. By this means the memory data-transfer rate can be very high. It remains for the cache memory organization to make adequate use of such multiword transfers.

When a request originates in the CPU for a new word, whether it be data or instruction, a check is made to see if it is already in the cache. If present, it is used directly; if not, a new access to main memory must be made. Since the cache is of limited size (16K bytes in the IBM 360/85), space must often be

CALCULATOR, DESK

sought to accommodate the new information. An algorithm based on history of use is used to identify the least necessary words for overwriting. Since the data in main memory is updated each time the CPU writes into the cache (a process called "store-through"), no data is lost in the overwriting process.

The search in the cache for the next word is made using an associative address store conceptually in the same way as for a general paging organization. In this associative search, the target word address is compared with stored addresses of words in the cache. The result of a match is the location in the cache of the desired word. No match initiates the main storage access process.

A general problem of updating of the cache and main store exists if, as is usual, the CPU and I/O do not both use the cache. This arises because, under these conditions, two versions of a variable might exist in two corresponding locations in cache and main store. Global and local solutions exist. The global solution is, as usual in a memory hierarchy, to restrict processing to blocks of data that are static with respect to I/O transfers; this insures that only consistent data is treated. This may be arranged by periodic programmed checking of flagged memory locations, modified by initiation and subsequent completion of I/O transfers. The second local solution is to insure that CPU modified data in cache is returned quickly and automatically to main store, as is done in the IBM 360/85.

K. C. SMITH AND A. S. SEDRA

CALCULATOR, DESK

For articles on related subjects see **CALCULATOR, ELECTRONIC**; and **DIGITAL COMPUTERS**.

For related biographical information see **LEIBNIZ, G. W.**; and **PASCAL, BLAISE**.

Man's needs for aids to calculation obviously began as soon as he began to count, assuming that human memories in those days were as bad as they are today. He used his fingers and perhaps toes (they were at least visible), and cut notches in a stick when a permanent record was required.

It was a natural development to replace fingers by small pebbles that would slide in a groove carved in a piece of wood. Such an elementary abacus was simplified when the pebbles were replaced by beads sliding on a wire or slim rod. There is good evidence

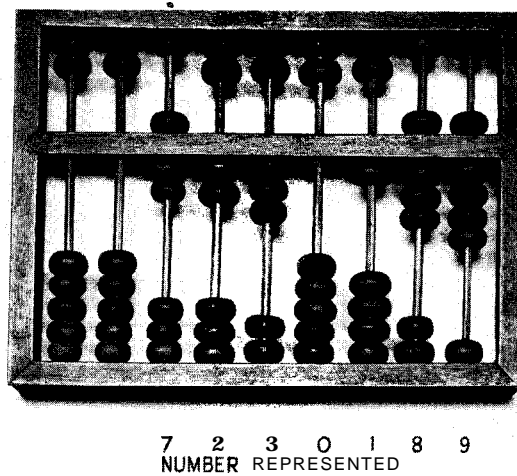


Fig. 1. A Chinese abacus. (Courtesy Science Museum, London. British Crown copyright.)

that the abacus was invented prior to 500 B.C.

The simplest abacus has either ten or nine beads on each wire, and the number system is obvious. In most modern forms of abacus (Chinese), the wire is divided into two parts and the coding system is essentially biquinary in that there are five beads below the division and two above. In the Japanese abacus, four beads are below and one above. The upper bead designates whether the rod represents more or less than 5; the lower one, whether 0 to 4 or 6 to 9.

Visitors to the Far East will know that the abacus is still a common form of desk calculator and is used with great dexterity and speed. It has been reported that a competition held in 1946 between an abacus and an electric calculator was (with human assistance!) easily won by the operator of the abacus.

The invention of anything resembling today's desk calculator had to await the development of a system of decimal notation as we know it today, and this did not occur until as late as the sixteenth century. One of the earliest aids was Napier's "bones" (about 1620), which effectively had multiplication tables written out on strips of bone or wood. Napier also invented logarithms. These greatly assisted in arithmetic calculation, at the cost of some accuracy, and are the basis of slide rule operations.

The mechanical calculator was first invented by Pascal, about 1640, and depended on linking a toothed gear wheel to a shaft and an arrangement for a "carry" from one wheel to its left-hand neighbor when the original wheel passed from 9 to 0. The accumulation gear wheels were driven by other

CALCULATOR, ELECTRONIC

machine could multiply directly, but this found little common application.

The twentieth century has seen a wide variety of desk calculators, all basically operating on a principle similar to that of Pascal's machine but with varying degrees of sophistication and aids to convenience. Input numbers may be set by moving selector levers to designated positions or, in some cases, by a simple keyboard. Operation may either be manual or electrical, and results are usually shown in plain figures on dials read through small windows (Fig. 2). There are also simple adding and listing machines that print results on a roll of tally tape (Fig. 3). Well-known manufacturers' names include Monroe, Marchant, Brunsvega, Facit, and a host of others.

G. J. MORRIS

CALCULATOR, ELECTRONIC

For articles on related subjects see **CALCULATOR, DESK**; and **INTEGRATED CIRCUITRY**.

Until the early 1960s desk calculators were essentially mechanical in operation, driven either by hand or an electric motor. They were bulky, usually fairly heavy, and certainly not easy to carry about. At their fastest, they took about a third- to a half-second to add a pair of numbers.

By the late 1940s technologists had found how electronics could be applied to computation, and by 1960 electronic computers were in wide use. The earliest models relied on the electronic tube as the basis of their circuitry, and therefore the machines were large. A significant reduction in size of the second generation of computers was made possible when the transistor replaced the tube. Circuits were, however, still far too big to use them in the construction of a desk-size calculator.

The solution to the problem was found in a by-product of the United States space research program. More than anything else, devices to be used in space vehicles had to be small and light, and vast amounts of development money were spent in search of ways and means of achieving this. One result was the microintegrated circuit, a method of compacting many electronic components and complicated circuitry in a very small space.

Transistors and other components in integrated circuits (IC) depend for their operation on minute chemical differences between adjacent parts of a tiny

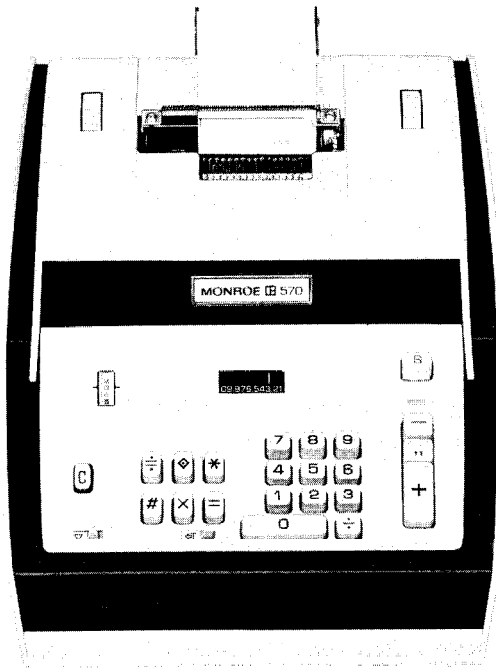


Fig. 2. The Monroe desk calculator with printed tally tape output.

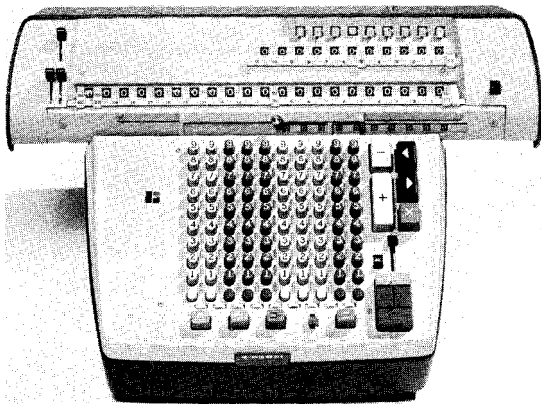


Fig. 3. The electromechanical Monroe 6F-212 calculator with output displayed but not printed (an obsolete model).

toothed wheels, set to represent a desired number and driven by a rotating hand crank. Such calculators, and they still exist today, were essentially adders and subtractors in which multiplication and division was performed by repeated additions and subtractions. In the 1670s Leibniz invented a much more complicated gearing arrangement by which a

CARD

flake of silicon. In the early days, a suitably treated wafer of silicon was chopped up to make many individual transistors. The major breakthrough to IC design was a process of making the desired interconnection between the components on the silicon wafer itself, and so reduce size and, even more important, cost.

These circuits are really tiny. An IC chip less than a quarter-inch square may contain over 7,000 interconnected transistors, together with diodes and other components. Such a massive array of electronic power in a small package (see Fig. 1) enabled designers of the desk calculator to replace mechanical parts with electronic components, and at one swoop reduce size and weight, greatly increase speed, and obtain silent operation.

The earliest electronic calculators appeared about 1962, and they have poured onto the market in great numbers since 1970. As production of the circuits and of the calculators themselves has rocketed, so prices have tumbled. In a very short time manufacturers have produced a bewildering array of machines from which to choose. The simplest models are already so cheap that a growing number of students use them to do homework.

Some electronic calculators are intended for desk use, but since 1970 numerous hand models have been available, some as small as a pack of cigarettes and easily carried in a pocket. Scientists, engineers, and mathematicians, as well as accountants and businessmen, use them extensively because their small size (Fig. 2) makes them extremely convenient. In these pocket calculators, numbers are entered by tapping a sequence of digitally marked keys; the desired arithmetic operation is ordered by tapping other keys marked with $+$, $-$, \times , \div , $=$, etc. Some models often cope automatically with decimal points. Results are usually shown in clear figures, using some form of lighted display. More elaborate hand-held machines are equipped with small aux-

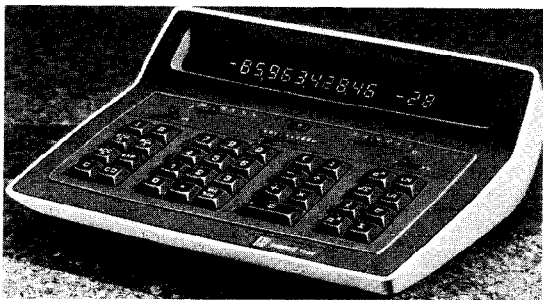


Fig. 1. The Monroe 1920 desktop electronic calculator.



Fig. 2. The assembled Sinclair "Executive" pocket calculator.

iliary memory capabilities, can handle constant multipliers, and some cases can evaluate trigonometric functions, square roots, reciprocals, etc., at the touch of a single button. Specifications of several models can include simple printing facilities.

G. J. MORRIS

CARD. See IBM CARD; and NINETY-COLUMN CARD.

CARD READING AND PUNCHING TECHNIQUES

For articles on related subjects see IBM CARD; and INPUT-OUTPUT DEVICES.

Punched card reading used to be performed electromechanically by sensing pins or reading brushes. The need for higher speed, reliability in reading, and lower cost resulted in the introduction of the photoelectrical reading technique, which is widely applied now. High reliability of reading is usually secured by checking techniques, for example:

1. A dual read station.
2. Echo (in which the data transmitted is returned to the point from which it was sent and compared with original data).
3. Validity.
4. Parity.
5. Single-access clutch.
6. Column strobe count.
7. Light/dark probe.

Usually a combination of these types of checks is used. Cards with readings that do not match are generally sent to a secondary stacker. In present-day

CARD READING AND PUNCHING TECHNIQUES

readers the scanning of a card is usually performed by columns (often referred to as "serial scanning") as opposed to the previously used scanning by rows (also called "parallel scanning").

Fig. 1 shows the principle of the photoelectrical reading technique. The reading of a card is performed at a dual read station consisting of two vertical columns of 12 photodiodes each and 2 gating diodes, located at each side of the read diode columns (these last two diodes are not shown in Fig. 1). Spacing between read diode columns is equal to that of one card column. Vertical spacing between diodes is 0.25 in. and the diodes span the 12 information rows of the card.

The two gating diodes are located between the horizontal rows of read diodes in such a way that they always see the solid portions of the punched card. Hence, these diodes are triggered only by the leading or by the trailing edge of the card.

All read station photodiodes are covered by a mask that contains rectangular holes, which are slightly narrower than the width of the punched card holes. Masking slots covering gating diodes 1 and 2 are smaller than those used for read diodes. The smaller dimension allows for a minute card skew or the possibility of slight disorientation between the information holes contained in the punched card.

The photodiode exciter source is a 28-volt, incandescent lamp. Light rays from the lamp are directed toward a periscopic mirror element, where

the reflective surface at the upper end directs light rays downward, through the optical glass, to strike the second reflective surface at the lower end. Parallel light rays are emitted from the edge of the glass and mirror element. The periscopic system distributes the light evenly over the 3-in. read station area.

When the punched card reaches the dual read station, parallel light rays pass through the information holes of the card and strike the corresponding read diodes. At the peak of the light transmission, timing circuits transmit a read gate probe, which permits information contained in the first card column to be recorded in the primary read register. Hence, each 12-bit column of the punch card is read in character serial mode. Once read and recorded, the first card column is read again by the second group of read diodes. This operation is performed in sequence for each column of the card. If any two information groups do not correspond, in the automatic mode of operation, a compare error signal is transmitted to the computer, and the computer may return a gate command signal to channel the card into a secondary tray. In the manual mode of operation, the compare error signal acts as a gate command to channel the card into the secondary receiving tray.

Card punching is performed at the punch station of a card punch. The punch station consists of a punch matrix and punch dies with punch mag-

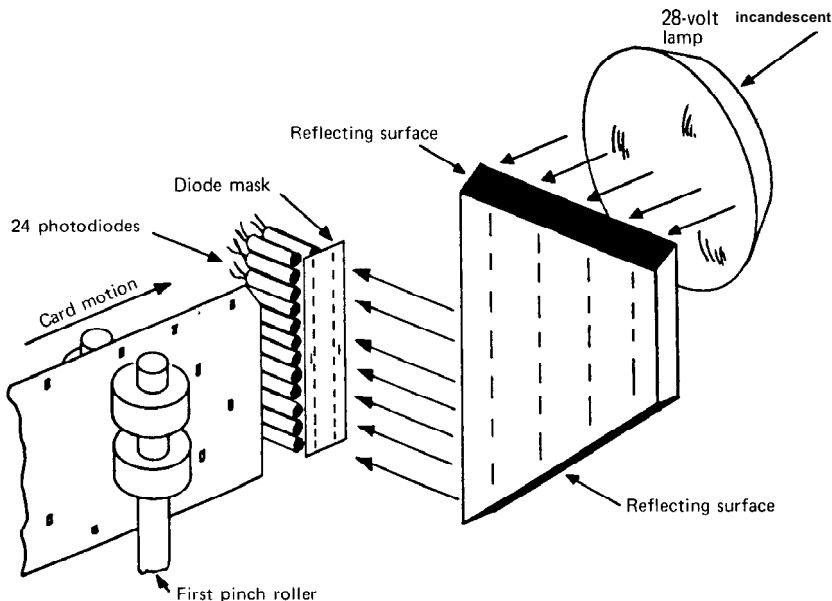


Fig. 1. The principle of a photoelectrical reading technique as used by Control Data Corp. in its CDC 405 high-speed card reader.

CATALOG

nets. A punch matrix has 80 holes corresponding with the 80 punching positions in a punch row of an 80-column card (if the punching is performed by rows) or it has 12 holes corresponding with the 12 punching positions of a punch column in an 80-column card (if the punching is performed by columns). The 80- or 12-punch dies are positioned upright with respect to the holes in the punch matrix.

Each punch die has its assigned punch magnet. These are set by the electronic image of the output information to be punched in the card. When activated, the punch die moves against the hole in the punch matrix, thus causing the punching of all data relative to a row (or column) of a card that is being moved (by rows or by columns) between the punch dies and the punch matrix.

The check of correct punching is generally performed by a "read-after-punch" verification,

echo, check on punching dies activated, and validity. Other error-checking capabilities generally built into the card punch include card synchronization and row-by-row hole-counting parity checking.

J. NECAS

CATALOG

For articles on related subjects **see FILES;**
and **MEMORY:** Auxiliary.

A catalog is a file, usually stored on some auxiliary storage medium, which contains an ordered list of names and other pertinent information on all files stored permanently in the computer system. In

FILE DIRECTORY LISTING

15		1	2	3	4
OWNER	FILE NAME	ED	C-DATE	E-DATE	L-DATE
JOB	GATHER-FILE	00	7-19-74	12-31-99	7-1 9-74
JOB	GATHERFILE	01	7-09-74	12-31-99	7-19-74
MSOS	FILE 54	00	7-19-74	7-19-74	7-21-74
MSOS	FILE 55	00	6-07-74	6-07-74	7-21-74
MSOS	FILE 56	00	7-19-74	7-19-74	7-21-74
MSOS	L-MSIO	00	12-22-72	12-31-99	7-21-74
NAD	NADSUB	21	6-03-74	6-03-74	7-19-74
NADS	SHORT CODES AND EXPANSION	00	7-18-74	7-18-74	7-19-74
PPR	INPFILE	00	6-07-74	6-07-74	7-21-74
PPR	PUNFILE	00	6-07-74	6-07-74	7-21-74
PPR	UTILFILE	00	1-30-73	12-31-99	7-21-74
RTS	ABSFILE	D6	6-07-74	12-31-99	6-07-74
RTS	IDFILE	00	12-22-72	12-31-99	5-02-73
RTS	LABELFILE	00	12-22-72	12-31-99	5-02-73
RTS	LIBDIRFILE	D6	6-07-74	12-31-99	6-07-74
RTS	LIBFILE	D6	6-07-74	12-31-99	6-07-74
RTS	MSDFILE	00	12-22-72	12-31-99	10-20-72
RTS	RESFILE	D6	6-07-74	12-31-99	6-07-74
TABLE	ORG CODES SORTED BY LEGAL ENT	01	7-06-74	7-06-74	7-11-74
TABLE	ORG CODES SORTED BY ORG CODE	01	7-06-74	7-06-74	7-11-74
TLOG	DAILY	00	7-21-74	7-21-74	7-21-74

Explanatory Notes

- 1 Edition number.
- 2 File creation date.
- 3 Expiration date.
- 4 Date file was last used.
- 5 Number of times file has been used since creation.
- 6 File size in disk segments.

Fig. 1. Example of an excerpt from a catalog. For purposes of finding a file, the file name consists of owner, file name, and edition for uniqueness.

addition to the name of the file (e.g., "1974-EARNINGS-RECORDS"), the catalog record may contain the creation date, an edition (or version) number, and a proposed (or assumed) expiration date. Normally it must contain the size of the file, the location of the file in storage (where the file has been stored on what device), and it will often maintain some usage measures such as number of accesses since creation, and date of last access, etc.

The catalog contains information on files stored on permanently mounted auxiliary storage units such as disks and drums. It also contains information on files stored on magnetic tape or on removable disk packs. When a tape or disk pack file is requested by a program, the information in the catalog together with other information is used by the operating system to determine if the file requested is available on line or if a tape or disk pack must be obtained and mounted. In the latter case,

appropriate instructions for the operator are issued at the computer console.

Files are stored permanently and cataloged when it is anticipated that they will be used repeatedly. In addition to data files, both source programs and object programs are often cataloged, the former so that they can be easily modified and the latter to enable repeated use without reloading the program from cards.

Fig. 1 contains an excerpt from a typical catalog of files showing the files stored on a particular disk pack.

The file catalog is also known by various other names among which the following are most common:

File Directory (as in Fig. 1)

File Name Table

Volume Table of Contents (for files on a

5	6	7	8	9	10	11	12	13	14
JSE CT	F- SIZE	B- SIZE	BLK CT	SEGCT	SEG	DT	DN	LSL	SL
3	20	256	1	1	1	854	8541	13792	20
39	20	256	13	1	1	854	8541	4736	20
294	50	480	282	1	1	854	8541	3216	50
8664	50	480	282	1	1	854	8541	22240	50
8665	80	960	277	1	1	854	8541	24640	80
14307	10	10240	4	1	1	854	8541	18640	10
217	3	960	9	1	1	854	8541	23520	3
8	6	504	43	1	1	854	8541	21568	6
445	50	1024	400	1	1	854	8541	25920	50
445	50	1024	240	1	1	854	8541	23040	30
2680	10	1024	0	1	1	854	8541	19808	10
	92	4	1462	1	1	854	8541	30960	92
	4	480	1	2	1	854	8541	240	2
0	40	392	36	2	1	854	8541	32	13
	3	500	17	1	1	854	8541	13744	3
1	241	960	962	1	1	854	8541	5072	241
	50	4	10	1	1	854	8541	272	50
	7	4	111	1	1	854	8541	19232	7
6	2	256	6	1	1	854	8541	32432	2
3	2	256	6	1	1	854	8541	13712	2
40	20	46	38	1	1	854	8541	19344	20

7 Block size in characters.

8 Block count, number of blocks in file.

9 Segment count; number of segments into which file is divided.

10 Segment number stored at this location.

11 Type model of disk device on which file is stored.

12 Device number.

13 Lowest segment location is track number on disk where this segment of file starts.

14 Length of this segment.

15 Since two people may use the same file name, an additional name is attached, giving the owner's name. In these cases the owner is the name of a package of programs (or system) which uses that file.

CELLULAR AUTOMATA

particular pack)

Permanent File Directory

C. L. MEEK

CELLULAR AUTOMATA

For article on related subject see **AUTOMATA THEORY**.

For article on related term see **PATTERN RECOGNITION**.

A cellular automaton is a theoretical model of a parallel computer, subject to various restrictions to make formal investigation of its computing powers tractable. All versions of the model share these properties: Each is an interconnection of identical cells, where a cell is a model of a computer with finite memory-i.e., a finite-state machine. Each cell computes an output from inputs it receives from a finite set of cells, forming its neighborhood, and possibly from an external source.

All cells compute one output simultaneously and each cell computes an output at each tick of a clock, i.e., after each unit time step. The output of a cell is distributed to its neighborhood and possibly to an external receiver.

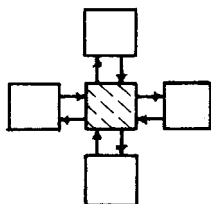


Fig. 1. A cell (hatched) and its neighborhood.

A version of the cellular automaton model exists for each set of choices in the following dichotomies: an infinite or a finite number of cells; a uniform interconnection scheme (all cells have neighborhoods of the same shape, e.g., that in Fig. 1) or a nonuniform scheme (Fig. 2); deterministic or **non-deterministic cells** (a choice of one output value at each unit time step or one of several values chosen randomly); the absence or presence of an external input (output), and in the case of an external input (output) the automaton is connected to all cells or to only a subset; Moore-type or Mealy-type cells (unit

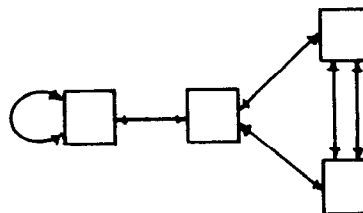


Fig. 2. A cellular automaton with nonuniform neighborhood.

time steps allowed or not allowed, respectively between inputs and the associated output); a static or dynamic interconnection scheme (neighborhood does or does not remain fixed in time). Some of the names associated with one or more of these versions are cellular automaton, tessellation automaton, modular computer, iterative automaton, intelligent graph, Lindenmayer system, and cellular network.

The first version of the cellular automaton, historically, was the cellular space obtained by selecting the first choice in each dichotomy above, but with no external input or output. It can be visualized in two dimensions as an infinite chessboard, each square representing a cell. It has been used to prove the existence of nontrivial self-reproducing machines, is capable of computing any computable function with only three states per cell and the four nearest cells as the neighborhood (Fig. 1), and can exhibit Garden of Eden configurations; i.e., patterns of cell states at one time, which can never arise in a given cellular space except at time zero. If an external input is assumed distributed to each cell, then the cellular space becomes what is usually called a "tessellation" space.

The cellular automaton is obtained from the cellular space by admitting only a finite, connected set of cells on the chessboard (Fig. 3). A cell with a neighbor missing has a special boundary signal substituted instead. The cellular automaton is **particularly** useful as a pattern **recognizer**, where the pattern comprises the states of the cells at time zero, especially if nondeterministic cells are allowed. A famous problem for the (deterministic) cellular automaton, the Firing Squad problem, calls each cell a soldier with one of them as the general-i.e., all cells but one are "off" initially-and asks if all soldiers can begin firing simultaneously by going into the

CENTRAL PROCESSING UNIT
(CPU)

For articles on related subjects see **ARITHMETIC-LOGIC UNIT; DIGITAL COMPUTERS; MEMORY; Main; and STORED PROGRAM CONCEPT.**

For article on related term see **OPERATING SYSTEMS.**

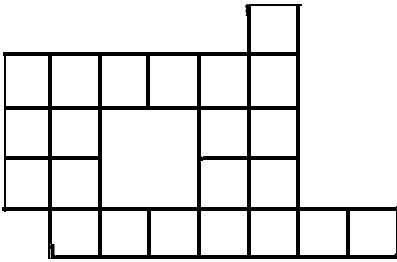


Fig. 3. A cellular automaton with uniform neighborhood of Fig. 1 assumed.

same state. The Firing Squad theorem, which solves this problem, guarantees an affirmative answer.

The Firing Squad theorem remains valid even when a nonuniform interconnection scheme is allowed. Thus, another version of the **cellular automaton**, the graphical cellular automaton (Fig. 2), requires only that the number of neighbors be fixed, not that they be in any fixed geometric relationship with a cell. They have been shown to be more powerful than the uniformly interconnected cellular automata.

The final type of cellular automaton to be mentioned, the dynamic cellular automaton, or Lindenmayer system, allows a cell to divide into daughter cells-regardless of the position of that cell in the initial array of cells-and allows the disappearance, or death, of cells. This version, with its dynamic interconnection scheme, is of interest to theoretical biologists as a model for the growth and development of living things.

If instantaneous communication is made possible between any two cells, by allowing Mealy-type cells, then each of the versions mentioned above gives rise to another. This class of cellular automata types is not well understood, although it is perhaps of the most interest in practical computing.

REFERENCES

- 1961. Hennie, F. C., III. *Iterative Arrays and Logical Circuits*. New York: M.I.T. Press and Wiley.
- 1968. *Cellular Automata*, ACM Monograph Series. New York: Academic Press.
- 1970. Burks, A. W. (Ed.). *Essays on Cellular Automata*. Urbana: University of Illinois Press.
- 1971. Gardner, M. "On Cellular Automata, Self-Reproduction, the Garden of Eden, and the Game Life". *Mathematical Games Department, Scientific American*, vol. 224, pp. 112-117.

A. R. SMITH

Although we still talk about "computers," some believe that the term "data processing system" is more descriptive of what is found in the normal computer room. The stress is on the term "system," implying that the modern computer consists of a selection of units of various types, all interconnected and **functioning** harmoniously with one another under central control. Most of the units in a system are called "peripheral" devices and serve either as the means of feeding raw data or file data into the system or of receiving results or updated files from the system.

The term "peripheral" conjures up a vision in which these units surround others, which serve as the focal point or center of the system (although this is rarely true physically). The name "central processor," or central processing unit (CPU), is used to describe elements that carry out a variety of essential data manipulations and controlling tasks at the heart of the computer.

Probably the most obvious element is the one required to carry out arithmetic and other operations on data, which is usually called the "arithmetic unit." It is designed to operate on a pair of numbers and carry out on them the processes of addition, subtraction, multiplication, and division. It can compare numbers and determine whether one is the greater or whether both are equal. These operations are carried out at very high speeds; even the slowest computers can do at least 10,000 such operations in a second, and the really fast "number crunchers" handle as many as 12 million.

The other obvious element is the control unit, required to supervise the functioning of the machine as a whole, calling into operation the various units as required by the program. It receives the program instructions one by one in sequence, interprets them, and sends appropriate control signals to the various units. It acts in many ways as a very sophisticated telephone switchboard operator, making **interconnections** between various parts of the system. When the control unit recognizes special signals (for example, that the result of a subtraction is negative), it

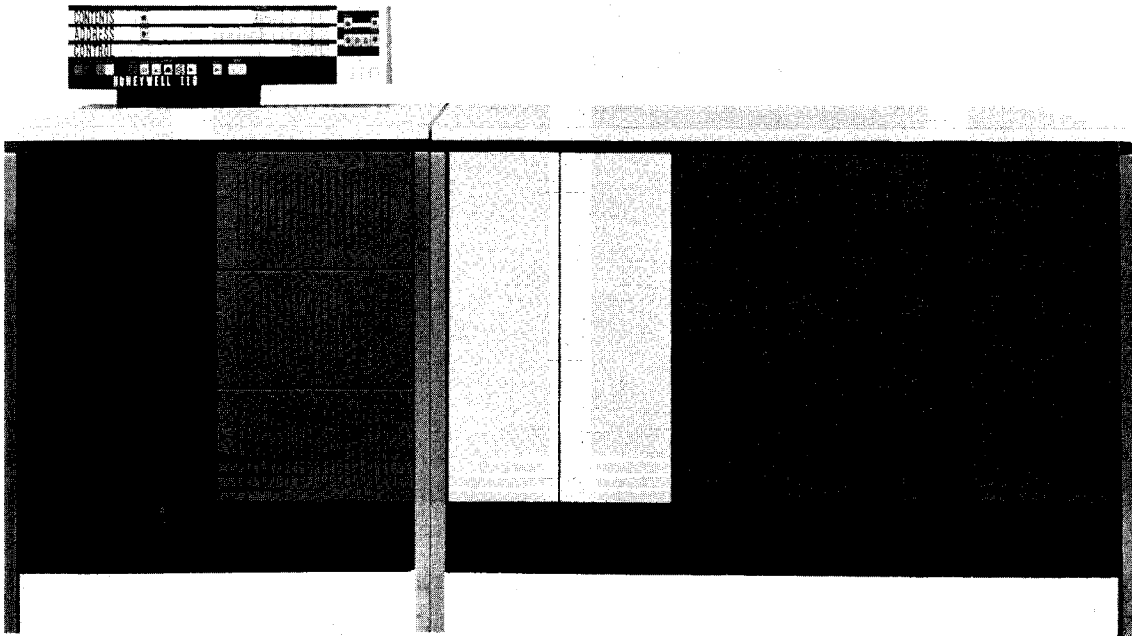


Fig. 1. Central processing unit of Honeywell 110 system.

can depart from the strict sequence of program instructions and jump to a different part of the program which is designed to deal with those circumstances.

Both the arithmetic unit and control unit depend heavily on the third main part of the central processor (Fig. 1), the main or central storage (or memory) unit. The arithmetic unit needs numbers on which to operate and needs to store intermediate results at some place until the end of the calculation. The control unit needs program instructions in rapid succession. Both data and instructions are held in memory. The program for a given job is read into memory from an input unit or auxiliary storage device as part of the setting-up procedure for the job. Data flows into memory from such devices as card readers and magnetic tape or disk units, and is manipulated while in storage to produce results that are output, for example, to a printer.

Memory is also used to store a complex of programs known as the "operating system"; this system is designed to supervise the total operation of the computer in as efficient a method as possible. These programs function in some ways analogous to "traffic controllers" as they have to monitor the flow of data around the computer, giving some streams right of way over others, opening up clearways for top priority messages, looking out for emergency

signals, and generally keeping things flowing smoothly.

The central processing unit is aptly named. It is very much at the center of computer activity, and it completes a massive amount of processing work both directly to produce the desired results and generally to supervise the efficient operation of the computer system as a whole.

G. J. MORRIS

CHAIN. *See* OVERLAY.

CHANNEL

For articles on related subjects *see* BUFFER; COMMUNICATION CONTROL UNIT; DATA COMMUNICATIONS; INTERRUPT; INPUT-OUTPUT DEVICES; MEMORY: Auxiliary; and MULTIPLEXING.

For articles on related terms *see* LOCKOUT; PRIVILEGED INSTRUCTION; and REGISTER.

Early Design. In the design of early computing systems it was usual to provide for only a minimum of input and output devices, such as paper tape or card readers and punches, and perhaps a line printer or teleprinter. All these peripherals were essentially slow. In such cases data could be transferred to and from the peripheral, character by character, and each unit had its special input or output line. Normally, data transferred between an I/O device and the store passed through the CPU. Later it was found necessary to provide many I/O devices. With the advent of magnetic tape units, a much faster device with a short crisis time (i.e., a need to be serviced very quickly if data was not to be lost), multicharacter block transfers became necessary.

In all cases, however, it was necessary to provide some indication of the status of the I/O device in use, such as "ready" or "busy." If a busy status of the device called upon was detected, the program usually had to stop and wait for the unit to become available again.

The need for block transfers to devices with short crisis times and the avoidance of delays due to unsuitable peripheral conditions led to the use of buffered peripherals and the development of continuously operating channels communicating directly with the store instead of through the CPU.

Autonomous Channel Operation. The eventual availability of fast buffered block peripherals such as magnetic tape and buffered peripherals such as card units and line printers called for the fast transfer of data to and from peripherals. If these transfers were controlled by the CPU, much time would be lost by the CPU, especially as character transfer was slow compared with other CPU operations. It follows that methods of autonomous transfer were needed. In these methods a whole block of data is transferred rapidly, word by word, to and from the main store, the cycles of the storage time taken for the word transfer being stolen from those available to the CPU. This usually causes only a slight hesitation of the CPU, whose storage cycle time of a 1-2 μs should be compared with that of magnetic tape unit, which usually operates at a rate of about 60 μs per word.

To facilitate block transfers directly between the store and the peripheral units, a controller called a "data channel" was introduced. There may be more than one channel. A data channel unit is essentially a small special-purpose computer. The CPU sends to the channel the length of the block of the continuous storage words to be transferred and the number of

those words to be transferred. The channel initiates the transfer, if possible; i.e., if the channel is not already busy and the channel equipment is available and ready to operate.

Usually the transfer from store to channel unit is in words, but the channel usually divides the words into a number of fields (or bytes), each of between 6 and 12 bits, suitable for acceptance by the peripheral unit controller and the peripheral itself. Then each byte is sent in turn, usually starting at the leftmost or the most significant byte. As a word is transferred to and from the store, the word count (the number of words still to be transferred) is decremented and the address incremented until the word count becomes zero after all data has been transferred.

In the case in which there is more than one channel, the channels are connected at the CPU end via a scanner device which may be called a "communication unit." The communication unit polls the various data channels in turn, and when a word is ready it is transferred, the data channel providing the address to store in or from which data is to be provided, and then providing or receiving the word. This scanning may be done sufficiently rapidly to avoid any crisis times, e.g., with magnetic tapes. The communication unit has direct access to the store and activates the input to the output from store. In some cases, the communication unit scans the channels in a defined order of priority. A communications unit may handle about eight channels.

The channels are connected to peripheral unit controllers. These may further break down the channel byte into units that can be handled by the actual I/O peripheral, e.g., a six- to eight-bit byte. The peripheral unit controller may not actually activate the output peripheral until its internal buffer has been filled by the channel, nor may it activate the input channel until its buffer has been filled by the peripheral device.

It is common for a read or write instruction relating to channel operations to be in two or more parts, since it is usually impossible to provide all data for the specification of the operation in one instruction word. Thus, the first part will specify and initiate the action of reading or writing, will give the address to which transfer will be made in the case of the rejection of the operation for any reason, and will contain the address of a "control word." The control word contains the length and head address of the block to be transferred and is that information which is actually passed to the channel unit. It is also possible to allow a sequence of data blocks to be transferred by providing a sequence or chain of

CHANNEL

control words that are sent one after another to the channel.

The control word also provides a function code that specifies and provides for certain types of variation of the normal mode of transfer of data such as skipping, reading, or writing zeros or terminating transfers, before or after the specified number of words indicated in the control word. However, the transfer of data is by no means the only function performed by a channel unit, for it may receive a variety of special orders from the CPU, such as channel and equipment selection and channel and equipment status inquiry.

Channel Capacity. The rate at which a channel can transmit data to or from an I/O device, or to or from main storage, is the channel capacity. This is usually given in bytes or kilobytes per second. The channel capacity must, of course, be great enough to service the fastest I/O device connected to it.

Computer manuals and channel specifications usually give figures for data transfer rates under the assumptions of ideal conditions. Actual data transmission rates are usually below these. If the channel hardware and the CPU hardware use the same registers, the channel may have to wait on the CPU for available registers (and vice versa), thus affecting transfer rates in a manner that cannot be determined a priori. The maximum rates given for discrete channels will also be lowered by the operation of other channels.

Since multiplexer or selector channels are essentially independent computers controlling I/O, they will, of course, have their transfer rates affected by the way they are programmed. If the data is entered into a contiguous area of storage, the rate of data transmission will be greater than if it is entered into a noncontiguous set of areas, where all sorts of addresses must be computed and the CPU notified as to which storage area is being affected. This use of noncontiguous memory for a data set is known as "data chaining." Of course, with data chaining, more areas of conflict with the CPU are possible, slowing either data transmission or processing.

Channel Command. A computer program is made up of a set of instructions that are decoded and executed by the CPU. Channel commands are instructions that are decoded and executed by the I/O channels. A series of commands in sequence constitute a channel program. Commands are stored in the main storage just as though they were instructions. They are fetched from main storage and are common to all I/O devices, but modifier bits are

used to specify device-dependent conditions. The modifier bits of the command may also be used to order the I/O device to execute certain functions that are not involved in data transfer, such as tape rewinding.

During its execution of a program the CPU will initiate I/O operations. A command will specify a channel, a device, and an operation to be performed, and perhaps a storage area to be used, and perhaps also some memory protection information about the storage area involved. All this information may appear in the command word, or the command may tell the channel in which locations in memory to seek the necessary information. Upon receipt of this information, the multiplexer channel will attempt to select the desired device by sending the device address to all I/O units (including controllers) attached to the channel. A unit that recognizes its address connects itself logically to the channel. Once the connection is made, the channel will send the executable command to the I/O device. The device will respond to the channel, indicating if it can execute the command. The channel will then make this information available to the CPU.

The I/O operation involving data transfer to or from a series of noncontiguous memory locations may involve a series of channel commands. Termination of an I/O operation involves channel-end and device-end conditions. These conditions are brought to the attention of the CPU via interrupts or programmed interrogation of the I/O device. The channel-end condition occurs when the data transmission is completed. The channel is considered busy until this condition is accepted by the CPU. The device-end signal is given when the I/O device has terminated execution of the operation. The device remains unavailable until it is cleared by the CPU.

Lockout, Cycle Steal, Hesitation. The memory of a computer cannot be accessed continuously, but only at specific points in time. The time elapsed between two consecutive points in time that the memory may be accessed by the processor or an I/O channel is known as a "memory cycle." The reason that the memory cannot be accessed continuously is that during a read/write operation, the information is not available, and some time must **elapse** before it is available again. Most memory cycle times are measured in microseconds or nanoseconds.

The CPU is essentially involved in processing the data that is in main memory while the channels are concerned with the flow of data between I/O

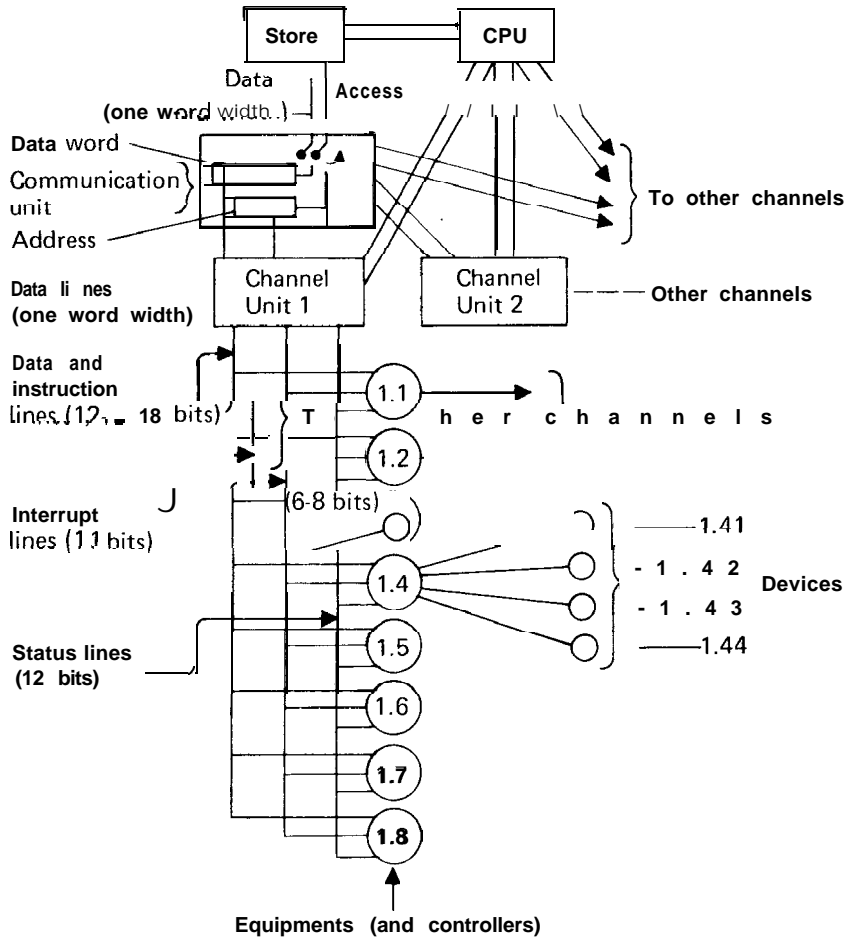


Fig. 1. Selector channel organization.

devices and main memory. Main memory is a high-speed data store, whereas peripherals are comparatively low-speed data stores. The channels and the CPU are busy moving data into and out of main memory. A source of conflict arises if they both need access to data at the same time. Since memory behaves the same way, whether the source or destination of the data is the I/O channels or the CPU, some method is needed to resolve the conflict.

Suppose another request for memory access is initiated by another channel while a memory cycle is going on. Since all requests must eventually be granted, but some more quickly than others, a priority system must be set up. It is the comparatively slow I/O devices, rather than the high-speed CPU, that must have their requests answered first. A tape speeding under the read head must give up its

data before the next data passes the read head; otherwise the data will be lost. The moving I/O devices must always have open space to accept more data and cannot be concerned with memory access problems. The CPU, on the other hand, goes from one stable state to another. Once the information is in its registers, it can wait. This will slow the processing, but will not lose any information. Therefore, a priority lock up is set up whereby the CPU is locked out from access to memory at the instant that the channels want to access the memory.

The memory cycle during which the channels have access to memory and the CPU is locked out is known as a "cycle steal," i.e., the channels have stolen the cycle from the CPU. For that cycle the CPU must stop and wait until it can access the memory again. This is known as "hesitation."

CHANNEL

Selecting a Peripheral. To select a particular peripheral unit, a special function code must initially be transferred to the channel, indicating the identity of the unit required. This is done by sending a "connect function" via the data line that the selected unit uses to make connection to the channel. Usually, a channel unit with a variety of equipments attached is connected to these equipments in what a communications specialist might call "multidrop manner"; all are connected by the same communication path to the channel (see Fig. 1). It may also occur that a particular peripheral equipment may be connected to a number of peripheral devices (e.g., a magnetic tape controller and a number of tape units) and that it may also be connected to more than one channel in case the alternate channel is already busy.

Setting a Peripheral. A special function code sent to a selected peripheral by its connected channel may specify operating conditions within which the external equipment is to operate or a condition in which an interrupt may occur, such as stopping the channel activity, selecting an interrupt on detection of a parity error, or stopping the operation.

Status. It is necessary in all multiprogrammed or time-shared systems to be able to detect the status of a channel and of the external equipment, the control word, and the control word address. Thus, the external equipment will provide a status code to indicate its operating condition. Depending upon the kind of equipment to which the channel is connected, certain bits in the code indicate that a parity error is present, or that a read or write is in progress, or that the operation is complete. Other codes in the status instruction cause the current data address and the word count to be sent to the CPU and/or the current control word address to be sent to other CPU registers.

In this way it is possible to detect not only the progress of a transfer and the chaining of control words but also a complex status, e.g., an empty card hopper, a card jam, empty line-printer paper, or a magnetic tape condition.

Detection of a busy channel or equipment status may be used appropriately to transfer control to a different program until a further interrupt recalls attention to the channel and its user program. An example is shown in Fig. 1 for one particular type of coding (e.g., CDC 3600).

Clear Channel. When initiating a program, or starting from a dead-stop condition or a re-

coverable difficulty, it may be necessary (1) to clear a channel by disconnecting all equipments from the specific channel and preventing any communication until a connect instruction is provided; or (2) to disconnect all units within an equipment (e.g., magnetic tapes on a multiple tape controller) and to clear the channel control words. The **CLEAR CHANNEL** instruction is also needed in case of difficulty with a channel operation and may be initiated by the operator.

Interrupts. Selecting an interrupt condition is performed by a function instruction that can select occurrences of address, and data and channel transmission parity errors. Associated with each channel, there is usually a special register in a channel unit which indicates the occurrence of one or more of these conditions. There is usually one bit in the register for each equipment condition. Additional bits are reserved for the use of the channel itself, such as an interrupt from the channel, channel data parity error, or control-word parity errors, as shown in Fig. 1.

However, most systems will operate in a normal or privileged mode; in the latter all interrupts are held inactive when processing an interrupt. The activity state of an interrupt can be set by a special function instruction that sets interrupts active, and returns the processor to the unprivileged state. The

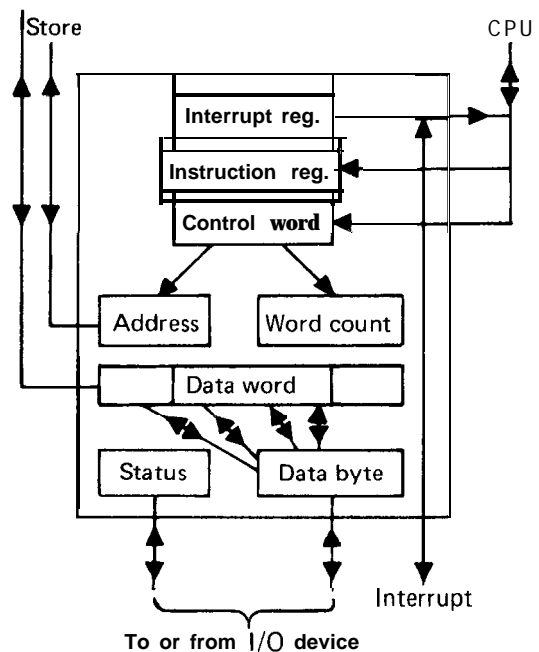


Fig. 2. Channel unit,

privileged status is automatically set on the detection of an interrupt when in normal state.

The structure of a channel unit is illustrated in Fig. 2. This shows the channel interrupt register, which indicates the conditions of the interrupt itself; the instruction register of the instruction which is received from the CPU; the control word, which gives the address and the number of words to be transferred; and the data word assembly registers, from which the data is to be sent from store. In addition to this, there is the status register, which is used to indicate the status of the connected unit or the channel.

Selector and Multiplexer Channels. Channels provide the ability to read, write, and compute concurrently. Each channel is essentially a small independent computer that responds to its own set of commands. The channel governs the flow of information between computer memory and the external world.

The two types of channels available are known as selector and multiplexer channels. High-speed I/O devices such as magnetic tapes, drums, and disks are usually connected to selector channels, while multiplexer channels usually have low-speed devices (card readers, line printers, teleprinters, paper tape readers) connected to them.

Numerous low-speed I/O devices connected to a multiplexer channel, may operate essentially simultaneously, their data being interleaved and directed to the proper locations in memory. Should high-speed I/O equipment be attached to a multiplexer channel, only one device will be able to

operate at a time because of the high transmission rates and short crisis times. The multiplexer channel is then said to be operating in "burst" mode. Selector channels always operate in burst mode. Usually, the operation of either type of channel does not inhibit processing.

A selector channel transmits information to memory from one I/O device at a time. A multiplexer channel may transmit information to memory from many I/O devices in an interleaved fashion. A selector channel will contain the information connecting the desired I/O device to an address in memory, a word count, and all other necessary information within the channel itself. Since a multiplexer channel may have as many as 256 sub-channels (connections between peripherals and the main memory), a table is set up in memory with the information necessary to each subchannel. During the actual transmission the information is in the multiplexer channel and is shuttled back and forth between multiplexer channel and main memory. Most computer manufacturers have this "table" area of memory set up so that it can be accessed only by the supervisory program, and not by the applications programmers. It is possible to access the memory for each subchannel transmission because memory access time is measured in microseconds, whereas access time of the slow I/O devices connected to a multiplexer channel is measured in milliseconds. It is therefore unnecessary to keep the control information for each subchannel in the channel between data transfers. A symbolic block diagram differentiating multiplexer and selector channels is shown in Fig. 3.

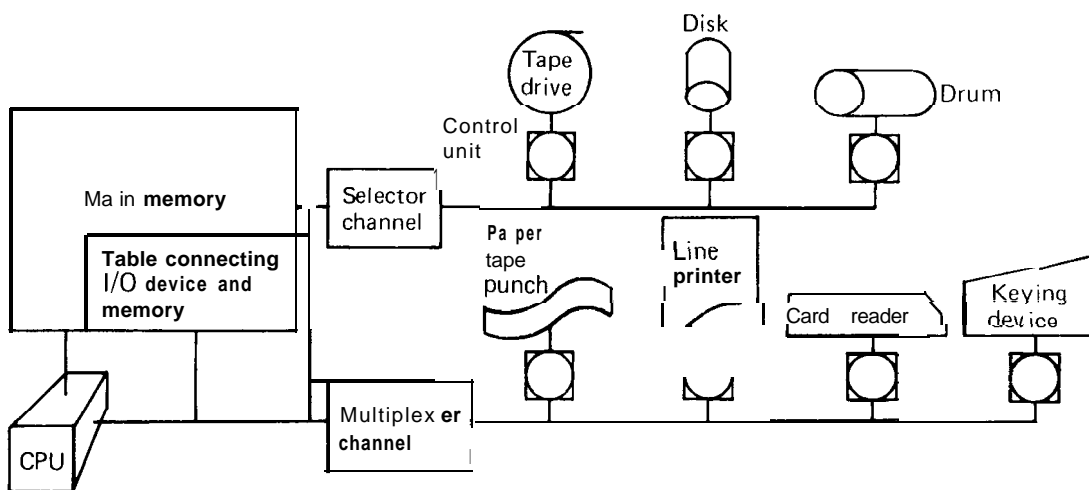


Fig. 3. A selector and a multiplexer channel,

CHARACTER SET

Communication Channels. Some channels may be devoted to communication between store or processor and a number of remote terminals. These terminals may be of the interactive type or of the batch type. In either case the channel makes connection via a multiplexer channel, and thence (usually) via public transmission lines of specified bandwidth prescribed by the data carrier organization. The communication outlets from the multiplexer are made via "data sets," which pass the data between the multiplexer and the transmission lines and convert signals to frequency modulation or otherwise adapt them to the communication line. Communication may be in one direction only (i.e., in "simplex" or "half-duplex" form, in which transmission may take place in either direction at any one time), or in "duplex" (needing four wires instead of two) in which transmission can pass in both directions at the same time.

Buffering. Buffering is used to gather information at a time when it is not needed so that the information will be available for processing when it is needed. An early use of buffering was to overlap I/O and CPU operations. For example, an I/O operation is initiated to read a block of data into memory. While the channels are controlling this operation, the CPU may go merrily on its way processing with available cycles. When the CPU needs the information from memory, it will be available and the CPU need not wait for the data.

Care must be taken during programming to see that the I/O is completed by the time the CPU needs the buffered information, and also that just enough information goes into each block of memory. Sometimes all I/O is directed toward a block of memory, a buffer, set aside just for I/O purposes by either the programmer or the supervisory program. The information is then moved to a working storage area for processing to take place; the I/O buffer area is then available for more I/O.

The concept of buffering has also found a significant application in peripherals. For example, in an on-line inventory control system, a storeman may type out his message and check it on a display device for errors and then transmit it to a computer. The message is stored in a small buffer in the remote terminal and then transmitted as a whole message to the main computer, rather than transmitting character by character as typed. In this manner the communications lines may be more efficiently used. These buffered I/O devices are the heart of all keying systems such as key-to-tape and key-to-disk. A card reader will have some buffer memory so that

the whole card may be read, stored, and then transmitted rather than read and transmitted one character at a time. Shrewd manipulation of buffers will greatly enhance the efficiency of its associated processing equipment by making necessary information available at the appropriate moment.

Current Trends. It is apparent that the complete function required of a channel allowing for multiprogramming and time sharing far exceeds those of simply reading and writing data. It is therefore becoming more common for channel units actually to be small programmed processors or minicomputers. This easily allows the extension of the channel functions; moreover, a greater variety of conditions can be specified by software design at a later stage of development. In this way a large CPU may be in charge of many small independent processors. This is one of the main design features of the CDC Cyber series of computers.

It is also becoming more common to provide a channel with an intermediary large buffer store to which the block may be rapidly transferred from the main store (e.g., at a 40 Mc rate), and from there transferred to the peripheral equipment connected to the buffer, at a slower rate appropriate to the equipment, while under the control of a small programmed processor.

T. PEARCEY AND M. PINE

CHARACTER SET

For articles on related subjects see COL-LATING SEQUENCE; and PROGRAMMING LANGUAGES.

A given set of symbols constitute the building blocks of any written language. For example, modern written English is composed of a character set that includes the so-called *alphabet* (A . . . Z) in its two forms (upper and lower case, or small and capital letters), the digits (0 . . . 9), some special punctuation marks such as comma (,), semicolon (;) etc., and the space character (). Using these elements, all written instances of the English language can be generated. However, the character set for English may not be sufficient for some other language such as (say) French in which additional characters must be added because of the use of accents.

Each computer language has its own character set which defines the set of characters that may be used in writing programs in that language. Table 1 gives the character sets of some common languages. PL/I has two possible character sets because it was designed at just the time when the IBM 026 keypunch (which had been the standard card-punching device and whose keyboard was limited to 48 characters) was being replaced by the IBM 029 keypunch, which allowed a larger character set, more convenient for writing programs in higher-level languages (but which has no provision for lower-case letters). As indicated in Table 1, implementations of

a language in a particular computer may not allow certain characters or may allow characters in addition to the officially defined ones.

Most of the characters in the character sets of higher-level languages have their usual meanings. But this is not always the case for the *special characters*, those which are neither letters nor digits. For example:

- * denotes multiplication
- = generally denotes replacement of the quantity on the left by the quantity on the right

Table 1. Character Sets of Fortran, PL/I, Cobol

Characters	Fortran	PL/I, 48-char. set	PL/I, 60-char. set	Cobol
A,B,C,...,Y,Z	X	X	X	X
0,1,2,...,9	X	X	X	X
blank	X	X	X	X
'	Z	X	X	X
@#%&.-?~			X	
> <			X	Y

Notes:
X indicates character is part of character set.
Y indicates character is part of character set, but is not allowed at many installations.
Z indicates character is not part of character set, but often is allowed.

One language whose character set is markedly different from all other higher-level languages is APL (*A* Programming Language) developed by Kenneth Iverson. APL contains many special symbols which serve as special *operations*. Some of these are shown in Table 2. In addition, APL defines many operators as a combination of two symbols from the character set. These are produced in practice by striking one key of a typewriter (with a special keyboard), backspacing, and then striking a second key.

Various codes have been developed to enable the characters used in higher-level languages to be represented by combinations of bits in a single 8-bit byte. Of particular note are the ASCII and EBCDIC codes.

Table 2. Some Special Characters in APL

Character	Name	Definition	Example (result of operator given under it)
⍒	Index	⍒ A generates indices in ascending order, starting from 1 to A	⍒6 1 2 3 4 5 6
⌈	Ceiling	⌈ B is the least integer greater than or equal to B	⌈14.7 15
⌊	Floor	⌊ C is the greatest integer less than or equal to C	⌊-5.9 -6
∘	Pi times	∘ D multiplies D by π	∘3 9.424777962
!	Generalized factorial	!E is the factorial of E when E is a positive integer (and the gamma funtion of E + 1 otherwise)	!7 5040

Chebyshev Approximation

References

1970. Ralston, A. *An Introduction to Programming and Computer Science*. New York: McGraw-Hill.
1972. Pakin, S. A *PL/360 Reference Manual*. Chicago: Science Research Associates.

J. A. N. LEE AND A. RALSTON

Chebyshev Approximation

For articles on related subjects see **APPROXIMATION THEORY**; **LEAST-SQUARES APPROXIMATION**; and **NUMERICAL ANALYSIS**.

Many computations on computers require the calculation of values of one or more functions such as square roots, sines, cosines, logarithms, **exponentials**, and other elementary functions or more complicated functions such as Bessel functions. Since computers can only perform the operations of arithmetic, these functions cannot be evaluated directly, but must be **approximated** by some other functions that can be evaluated arithmetically. For example, a common method for computing the square root of a number A is the following application of the Newton-Raphson method:

$$x_{i+1} = \frac{1}{2} \left(x_i + \frac{A}{x_i} \right) \quad i = 0, 1, 2, 3, \dots \quad x_0 = A$$

It can be shown that x_i gets arbitrarily close to A as $i \rightarrow \infty$. For example, let $A = 2$. Then

$$\begin{aligned} x_0 &= 2 \\ x_1 &= 1.5 \\ x_2 &= 1.41666 \dots \\ x_3 &= 1.414215 \dots \end{aligned}$$

while $\sqrt{2} = 1.414213 \dots$

The general problem we wish to consider here is: Given a function $f(x)$ and an interval $[a, b]$ on which we wish to approximate $f(x)$, find an approximation to $f(x)$ on this interval-which can be computed arithmetically-of minimum error. But what do we mean by minimum error? In many problems in mathematics this would mean minimum least squares error over the interval. But in approximating functions for computers we are more usually interested in minimizing the maximum error

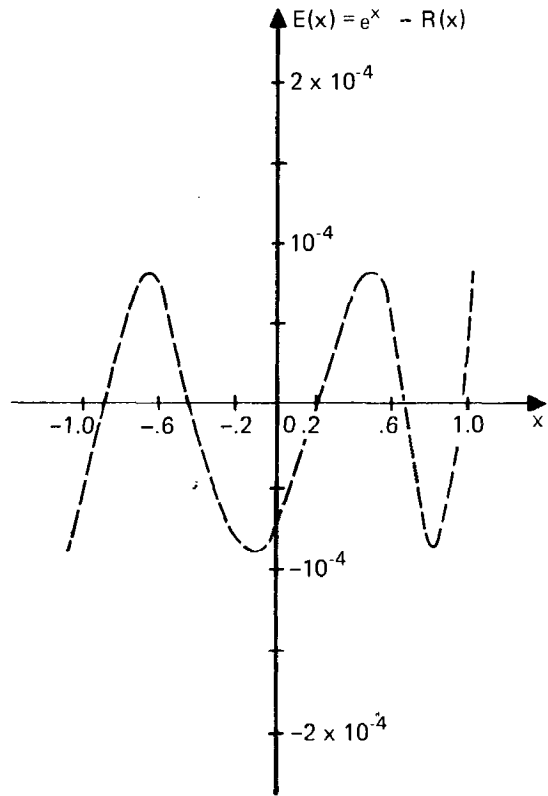


Fig. 1. Error in Chebyshev approximation to e^x on $[-1, 1]$ as a ratio of two quadratics.

on the interval, for then the user of the approximation always knows that the worst possible case is as favorable as it can be. Rigorously stated, we wish to find an approximation $R(x)$ which has the property that

$$r = \max_{[a,b]} |f(x) - R(x)|$$

is smaller than for any other approximation. Such a **minimum-maximum error approximation**, or **minimax** approximation, is usually called a "Chebyshev approximation" after the great Russian mathematician P. L. Chebyshev (1821-1894), whose name is transliterated from the Russian in a variety of other ways (e.g., Tchebycheff).

The question remains of what form $R(x)$ should have. If it is to be evaluated arithmetically, then the most general function it can be is a **rational function**, i.e., the ratio of two polynomials. For example, the Chebyshev approximation to the exponential function e^x on the interval $[-1, 1]$, which is the ratio of

two quadratic polynomials, is given by

$$R(x) = \frac{1.00007255 + 0.508636 \, 18X + 0.08582937x^2}{1.0 - 0.49109193X + 0.07770847x^2}$$

for which $r = 0.86899 \times 10^{-4}$. The error, $E(x) = e^x - R(x)$, is shown in Fig. 1. It exhibits the characteristic property of Chebyshev approximations of alternating between its greatest and least values twice more than the sum of the degrees of numerator and denominator of $R(x)$ or, in the example above, $2 + 2 + 2 = 6$ times.

REFERENCE

1965. Ralston, A. *A First Course in Numerical Analysis*. New York: McGraw-Hill.

A. RALSTON

CHECKPOINT AND RESTART

For articles on related subjects see **COMPUTER**, **USING A**; and **DEBUGGING**.

A designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at some arbitrary time in the future is called a "checkpoint."

The primary purpose of a checkpoint is to avoid repeating the execution of a program from its beginning, should an error or malfunction occur somewhere in the middle of processing. This is especially effective in runs involving several hours of machine time. For such situations it is often appropriate to set up checkpoints at a number of strategic places in the program, either with all checkpoint information being saved, or by using a less conservative system in which the information captured at the most recent checkpoint replaces (overwrites) the previous set. Then, should difficulties arise, it is possible to take corrective action and resume processing from the last checkpoint, rather than starting over. Since the manipulations associated with checkpoint/restart procedures can consume substantial amounts of time and storage, it is possible to have situations in which it is more economical to avoid checkpoints. This capability is implemented by means of a procedure (often termed a "checkpoint routine") that captures the status of the

program at the particular instant when it stopped and copies it onto an auxiliary storage medium. This data includes the contents of the special registers, storage locations associated with the program, and other information relating to the status of input/output devices. Later on, another procedure (a restart routine) can reset the system to resume processing by reading in and restoring the checkpoint information.

In many systems the checkpoint and restart routines are prepackaged software components accessible to the higher level language programmer via ordinary **CALL** statements. These facilities usually include numerous options that allow the programmer to exercise some control over the type of information gathered, the form in which it is stored, and the circumstances under which the restart is to proceed.

The introduction of complex operating systems has prompted an expansion in the use of checkpoint-restart procedures beyond the context of insurance against malfunctions. Depending on the strategy implemented in a particular system, it may be decided to interrupt a particular run, releasing its storage for other purposes with the intent of resuming that run at some later (presumably more propitious) time. In order to handle that type of procedure without the user's involvement, the checkpoint/restart process must become completely automated.

S. V. POLLACK

CIRCUITRY. See **COMPUTER CIRCUITRY**; and **INTEGRATED CIRCUITRY**.

CLOSE AND OPEN A FILE. See **OPEN AND CLOSE A FILE**.

CLOSED SHOP. See **OPEN SHOP**.

CM. See **COMPUTER-MANAGED INSTRUCTION**.

COBOL. See **PROCEDURE-ORIENTED LANGUAGES**; **PROCEDURE-ORIENTED LANGUAGES, PROGRAMMING IN**.

CODASYL. See CONFERENCE ON DATA SYSTEMS LANGUAGES.

CODES

For articles on related subjects see **BINARY CODED DECIMAL, NATURAL; ERROR CORRECTING CODE.**

For articles on related terms see **ASCII; COLLATING SEQUENCE; COMPLEMENT; EBCDIC; HOLLERITH, HERMAN; and PARITY.**

A code is a correspondence between a symbol of an alphabet (e.g., our alphabet of letters) and a number of digits of a number system (e.g., six bits for base 2). To be more precise, the mathematician would say that a code is a couple, which might be represented as $\Gamma(\Sigma, \Pi)$. Here, Σ is the symbol space and the Π are numeric combinations. Suppose that S is some symbol in the symbol space Σ and P is one of the permutations of the digits in a numeric counting system Π . We might say "S is mapped into P" or that "S is represented by P," using the following symbols:

$$S \rightarrow P \quad \text{or} \quad S \equiv P. \quad (1)$$

I prefer to call P a "combination." Since P consists of n digits, it can be written as

$$P = P_1 P_2 P_3 \dots P_n, \quad (2)$$

where P_i is any digit of the counting system with base B ; i.e.,

$$P_i = 0, 1, 2, \dots, \text{or } B - 1. \quad (3)$$

To make this more concrete, let us examine the case where the symbol space Σ_A consists of letters of the alphabet. Let each combination P consist of two decimal digits, $B = 10$ and $n = 2$. A very simple code might assign numbers consecutively to the letters so that we would have

$$A \equiv 01, \quad B \equiv 02, \quad C \equiv 03, \dots, Z \equiv 26. \quad (4)$$

It is convenient here to introduce the number operator ν , whose action is to find the number of elements in a set. Notice for our example that

$$\nu \Sigma_A = 26 \quad \text{and} \quad \nu \Pi = 100 \quad (5)$$

Since there are many more permutations than there are symbols in the symbol space, it is customary to find many permutations that go unassigned. These are sometimes called "forbidden combinations."

Need. Data is an abstraction of information in the real world. People keep this information in the form of symbols. The computer stores information in the various hardware elements that constitute it. Elements have been designed which have two or more states. An element such as the Nixie tube has ten states. But by far the most common, least expensive, and most efficient element is the *bistable* device; it has only two stable states. For the computer to represent information, it must be structured so that the devices used in the computer can accommodate it. Since there are not enough states in a single bistable device to represent each symbol as a human being uses it, the symbols are represented by a combination of these settings, by a combination of some codes.

It might seem initially that any representation of a symbol would do. This is not so. The design of a code usually must take into account the following requirements:

1. The original order relations that apply to the symbols within the symbol space should apply to the relation between combinations in the code.
2. Operations applied to the symbols should have analogous operations, which-when defined upon the combinations-produce a corresponding result.
3. The representation should be efficient (to minimize the number of combinations that go to waste) and the digit string should not be too long.

Decimal Codes. A decimal code provides a representation for the decimal numbers. The codes of interest to us use the base 2. Hence, these are called "binary coded decimal codes" (BCDs). To summarize their characteristics

$$\nu \Sigma_D = 10, \quad B = 2 \quad n \geq 4. \quad (6)$$

Note that these codes can be four bits or more. There are many useful codes that consist of more than four bits, and it is an error to believe that BCDs are *all* four bits; they are not.

There are several means for associating the symbols with combinations:

1. *Weighted codes* assign different weights to each bit in the combination as discussed shortly.
2. *Transition rules* may be created to indicate how the code for the successor number is created from the code for any given number.
3. Finally, there is the explicit assignment, where no rule as such exists.

As an example of a random BCD we have

$$0 \equiv 111000, \quad 1 \equiv 101010, \quad 2 \equiv 110011, \quad \text{etc.} \quad (7)$$

Four-Bit BCDs

WEIGHTED CODES. Let us label the bits of the combination that represents a decimal digit. Unlike Expression (2), where the subscripts go from left to right, we will now order the subscripts 1 through 4 in reverse, going from right to left. Thus, if D is a decimal digit, then we have

$$D \equiv b_4 b_3 b_2 b_1. \quad (8)$$

A weighted code associates a weight with each bit and might be stated symbolically as

$$b_i \longleftrightarrow W_i \quad i = 1 \text{ to } 4 \quad (9)$$

The requirement of the weighted code can be stated in the form of an algorithm: Multiply each bit by its weight and then total these. The total must be equal in value to the digit. Stated symbolically, we have

$$D = \sum_i b_i W_i = b_4 W_4 + b_3 W_3 + b_2 W_2 + b_1 W_1 \quad (10)$$

Some restrictions arise in setting up the weights:

1. For each digit to be encoded, there must be a combination of bits and their corresponding weights, whose total-using Expression (10)-is equal to the value of the digit.
2. When two combinations exist which, when substituted into Expression (10), yield the same digit, D , then another rule must be provided to decide which combination will be used.

8421 Code. The weighted 8421 code is illustrated in Table 1. From left to right, weights 8, 4, 2, and 1 are assigned to the bits that make up the combination. When the bits are set to 0 or 1, the

	Table 1	Table 2	Table 3	Table 4
Weights Digits	8 4 2 1 C o d e	7 4 2 1 C o d e	7 4 2-1 C o d e	Excess-3 C o d e
0	0 0 0 0	0 0 0 0	0 0 0 0	0 0 1 1
1	0 0 0 1	0 0 0 1	0 0 1 1	0 1 0 0
2	0 0 1 0	0 0 1 0	0 0 1 0	0 1 0 1
3	0 0 1 1	0 0 1 1	0 1 0 1	0 1 1 0
4	0 1 0 0	0 1 0 0	0 1 0 0	0 1 1 1
5	0 1 0 1	0 1 0 1	0 1 1 1	1 0 0 0
6	0 1 1 0	0 1 1 0	1 0 0 1 (0110)	1 0 0 1
7	0 1 1 1	1 0 0 0 (0111)	1 0 0 0	1 0 1 0
8	1 0 0 0	1 0 0 1	1 0 1 1	1 0 1 1
9	1 0 0 1	1 0 1 0	1 0 1 0	1 1 0 0
*(A)	1 0 1 0	1 0 1 1	1 1 0 1	1 1 0 1
*(B)	1 0 1 1	1 1 0 0	1 1 0 0	1 1 1 0
*(C)	1 1 0 0	1 1 0 1	1 1 1 1	1 1 1 1
*(D)	1 1 0 1	1 1 1 0	----	----
*(E)	1 1 1 0	1 1 1 1	----	----
*(F)	1 1 1 1		----	----

* Forbidden combinations.

resulting number is shown in the left column of the table.

The six entries at the bottom of the table provide values 10 through 15. Of course there are no digits to correspond to these values in the decimal system. Hence, these combinations are forbidden. If these occur, the computer should signal an error.

Note that these combinations would be legal if the base for our system were 16. Hence, we will return to this table when we discuss the hexadecimal (base 16) system.

Finally note that the sequence of combinations for the 8421 code is the same sequence in which these binary numbers occur in the binary counting system. The binary counting system has been called the "natural" counting system in the literature. Hence, the appellation *natural binary coded decimal*, or simply NBCD for the code of Table 1.

7421 Code. Table 2 presents the 7421 code. Again there are six forbidden combinations. The bits that constitute each combination are calculated so that Expression (10) will yield the digit value.

A problem arises for encoding the digit 7. There are two combinations, 1000 and 0111, both of which yield the value 7. An auxiliary rule is required to settle this difficulty: Use the combination with the least number of 1's in it (i.e., 1000).

742-1 Code. The code for these weights is presented in Table 3. It illustrates that one or more of the weights may be negative as long as the weights

CODES

fulfill the requirement that all digit values must be created. This time we find that there are two combinations that yield the digit value 6. Since both have the same number of 1's, we choose the combination with the 1 in the least significant place.

XS 3 Code. To show that not all codes require weights explicitly, we examine the XS 3 (excess 3) code presented in Table 4. The rule for generating this code requires that we use the NBCD code for a digit, call it n_D , and add the binary number 0011 (i.e., 3 in decimal) to it:

$$D \equiv n_D + 0011 \quad (11)$$

What use would there be for this code? It has two advantages :

1. No proper combination consists of all zeros; therefore no combination will be mistaken for a null transmission.
2. It is a self-complementing code.

A self-complementing code is very valuable because it possesses this quality: The combination for the complement of a digit is the complement of the combination for that digit. The complement of a number is needed when we do subtraction by addition and complementation. For decimal arithmetic this requires that we subtract the value of the digit from 9. In our binary code the complement of a combination b_D is taken with respect to the largest valued combination; for a four-bit code, this would be 1111. Then, our definition for a self-complementing code is one for which the following holds:

$$9 - D \equiv 1111 - b_D \quad (12)$$

As an example of how XS 3 fulfills this requirement, we have

$$\begin{aligned} 2 &\equiv 0101, & 9 - 2 &= 7, \\ 1111 - 0101 &= 1010, & 7 &\equiv 1010 \end{aligned} \quad (13)$$

Natural Binary Coded Hexadecimal.

This code (NBCH) is what others have called simply the "hexadecimal code." Programmers normally deal with NBCH using bytes. The byte consists of eight bits or two halves, each consisting of four bits. If the combination for each half has a different symbol to represent it, then this simplifies the description. The binary values with decimal equivalent between 10 and 15 have been assigned the upper case letters A through F as shown in Table 1 in parentheses. Thus,

	Table 5 2-out-of-5 Code	Table 6 Biquinary Code	Table 7 MBQ Code	Table 8 Gray Code
Weights Digits	74210	50 43210	542 1	
0	11000	01 00001	0000	0000
1	00011	01 00010	0001	0001
2	00101	01 00100	0010	1001
3	00110	01 01000	0011	1101
4	01001	01 10000	0100	0101
5	01010	10 00001	1000	0111
6	01100	10 00010	1001	1111
7	10001	10 00100	1010	1011
8	10010	10 01000	1011	0011
9	10100	10 10000	1100	0010
10	----	-----	----	1010
11	----	-----	----	1110
12	----	-----	----	0110
13	----	-----	----	0100
14	----	-----	----	1100
15	----	-m--we-	----	1000

the programmer can describe the byte consisting of 10110101 in hexadecimal as B5.

Other BCDs. If we do not restrict ourselves to four-bit BCDs, we can provide one or more of the following advantages :

1. Error detection.
2. Simplicity of combination construction.
3. Simplicity of implementation in hardware.

Z-out-of-5 Code. The 2-out-of-5 code provides the first advantage and is illustrated in Table 5. Every five-bit combination that represents a digit contains exactly two 1's. Since there are ten such permutations, this works out well. To assign each combination, we establish a set of five pseudo-weights. One of these weights, W_1 , is 0, and the bit corresponding to this weight, b_1 , should be set to 1 when the value of the digit being encoded corresponds to one of the nonzero weights; this is true for the digits 1, 2, 4, and 7. The weights work out for all digit values except 0; this digit uses bits with weights 7 and 4, which obviously do not sum to 0; hence, the term "pseudo-weights."

Biquinary. The biquinary code is a seven-bit code using exactly two 1's; it is illustrated in Table 6. One of the 1's is chosen from the left two bits; the other is chosen from the right five bits. The weights are used as would be expected. This code provides error detection whenever more than one "1" appears in either half of a combination, and also provides a

logical progression from one combination to the next, which is useful for implementing arithmetic.

MBQ Code. The modified biquinary code (MBQ), illustrated in Table 7, is derived from biquinary by replacing the first two bits by a single bit, and the last five bits (which represent 0, 1, 2, 3, or 4) by three bits, which represent them in NBCD.

Gray Code. The Gray code, invented and patented by F. Gray, was developed to fill a particular requirement. Many of the devices designed to read information into the computer depended on the mechanical position of a shaft. Attached to the shaft was an encoder that produced electromechanical or optical signals corresponding to the shaft rotation. This created a transition problem. From Table 1 it is clear that the combination for 7 is 0111 and that for 8 is 1000. As the shaft rotated, the apparatus for reading out the position could not be depended upon to change simultaneously in each bit position. Thus, totally erroneous readings occurred. If b_4 goes to 1 before the other bits change in going from 7 to 8, the output would be read as 15.

To overcome the transition difficulty, a code was devised whereby successive combinations changed in exactly one-bit position only, as shown in Table 8. The Gray code is not the only one that does this, but it has long been a standby. The length of each combination L is a function of the number (N) of discrete shaft positions to be encoded, as given by the formula

$$2^{L-1} < N \leq 2^L \quad (14)$$

Fig. 1 displays a code disk for the Gray code of Table 8, which indicates the change of one bit at a time.

Full Alphabet. Thus far we have restricted our symbol set Σ to decimal numerals. As computers went from infancy to early childhood, it was obvious they could be applied to accounting problems in which alphabetic output is mandatory, and where we encounter the following classes of symbols :

1. Letters: the alphabet from A to Z.
2. Numerals 0 through 9 (which we have already examined).
3. Punctuation.
4. Special symbols, which include &, @, \$.

The question arose of how large or how small the symbol space should be. With six bits we can

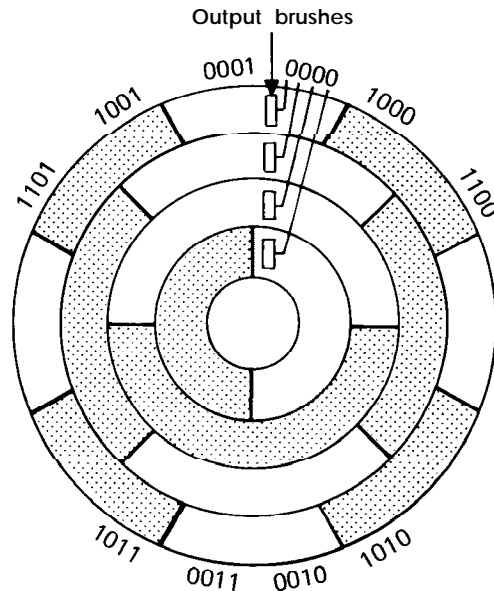


Fig. 1. The Gray code disk for Table 8.

encode 64 symbols. Since most printers today have a 48-character set, why should we want more?

Perhaps the manufacturers have convinced industry that eight bits are really necessary because they need to represent lower-case letters, special symbols, control characters, and space to accommodate future requirements. This argument seems unconvincing. An estimated 30% inefficiency has been introduced into computers because of this standardization.

Hollerith. The Hollerith card code for IBM cards (which enables each column in the card to represent alphabetic, numeric, or symbolic information) is discussed in the article IBM Card.

IBM 1401. When the second generation IBM 1401 computer was developed, it was intended to replace electronic accounting machines that rely entirely on punched cards. Therefore, as expected, the code used in the 1401 computers, as given in Table 9, corresponds closely to the Hollerith code. We note the following:

1. Digit bits 1 through 9 in the 8421 columns are given by NBCD, but 0 is represented by 1010.
2. The bits B and A represent the three zone punches 12, 11, and 0 as follows: $BA = 11$ for 12, $= 10$ for 11, $= 01$ for 0, and $= 00$ for no zone punch.
3. The C bit is the check bit (or parity bit,

CODES

	Table 9	Table 10	Table 11
	IBM 1401		
Numeric/ Alphabetic	Code C B A 8 4 2 1	EBCDIC in NBCH	ASCII-8 in NBCH
A	0 1 1 0 0 0 1	C1	A1
B	0 1 1 0 0 1 0	c2	A2
C	1 1 1 0 0 1 1	c3	A3
D	0 1 1 0 1 0 0	c4	A4
E	1 1 1 0 1 0 1	c5	A5
F	1 1 1 0 1 1 0	C6	A6
G	0 1 1 0 1 1 1	c7	A7
H	0 1 1 1 0 0 0	C8	A8
I	1 1 1 1 0 0 1	c9	A9
J	1 1 0 0 0 0 1	D1	AA
K	1 1 0 0 0 1 0	D2	AB
L	0 1 0 0 0 1 1	D3	AC
M	1 1 0 0 1 0 0	D4	AD
N	0 1 0 0 1 0 1	D5	AE
O	0 1 0 0 1 1 0	D6	AF
P	1 1 0 0 1 1 1	D7	B0
Q	1 1 0 1 0 0 0	D8	B1
R	0 1 0 1 0 0 1	D9	B2
s	1 0 1 0 0 1 0	E2	B3
T	0 0 1 0 0 1 1	E3	B4
U	1 0 1 0 1 0 0	E4	B5
V	0 0 1 0 1 0 1	E5	B6
W	0 0 1 0 1 1 0	E6	B7
X	1 0 1 0 1 1 1	E7	B8
Y	1 0 1 1 0 0 0	E8	B9
Z	0 0 1 1 0 0 1	E9	BA
0	1 0 0 1 0 1 0	F0	50
1	0 0 0 0 0 0 1	F1	51
2	0 0 0 0 0 1 0	F2	52
3	1 0 0 0 0 1 1	F3	53
4	0 0 0 0 1 0 0	F4	54
5	1 0 0 0 1 0 1	F5	55
6	1 0 0 0 1 1 0	F6	56
7	0 0 0 0 1 1 1	F7	57
8	0 0 0 1 0 0 0	F8	58
9	1 0 0 1 0 0 1	F9	59

which is discussed in the following section, "Error Detection and Correction").

There are many other six-bit codes that are characteristic of the machine that employs them. They are generally listed in an appendix of the programmer's manual for the machine.

EBCDIC. The Extended Binary Coded Decimal Interchange Code (EBCDIC) is an eight-bit code developed by IBM and is available on all IBM 360 and IBM 370 computers. In fact, it is the only code used with the IBM 370. **NBCH** may be used to convey each combination, as shown in Table 10. Thus, A is represented by C 1, which in turn means

11000001. A more complete discussion of EBCDIC appears elsewhere in this encyclopedia.

ASCII. The American Standard Code for Information Interchange (ASCII) is actually a seven-bit code. To make it an eight-bit code, it has been embedded into ASCII-8, a comparable eight-bit code. Table 11 displays the encoding of the important characters-letters and numerals. A more complete discussion is found elsewhere in this book.

Contrast. There is no clear superiority of either EBCDIC and ASCII-8, but there is an important difference. The collating sequence for EBCDIC has the numerals follow the letters; for ASCII-8, the reverse is true. Hence, documents coded and sorted under one system would be in a different order than if they were coded and sorted by the other.

Error Detection and Correction. In the case of biquinaty, we have seen how a code can be constructed with error detection properties. This is helpful, and even necessary, in many situations, such as:

1. Information is transmitted from one site to another along lines where noise or other signal distortion might occur.
2. The data is recorded on a medium that is not impervious to noise so that l's may get lost and be read as O's, or O's may be interpreted as l's because of noise.
3. Devices within the computer may become faulty and create or destroy information,

Parity. The simplest means for detecting errors is to attach an extra bit to each combination of the code, called a "parity" bit. This bit is set to 0 or 1, according to the scheme used: For *odd* parity, the total number of l's, including the parity bit, must be odd; for *even* parity, the total number of l's, including the parity bit, must be 0 or even. An example of the use of an odd parity bit (also called a "check" bit), labeled C, is shown in Table 9. There are two phases in the use of the parity bit: creation and checking.

In the **creation phase**, the combination is examined and a parity bit is created so that the number of l's in the total combination is proper. Now the combination can be transmitted from one place inside or outside the computer to another place. When it arrives there, the checking action follows. Circuitry similar to that for parity creation examines the combination exclusive of the parity bit as though it were creating that parity bit. If this developed bit and the accompanying parity bit coincide, a single

bit error could not have occurred, and the information is **accepted**.

Other Codes. Many different kinds of computers have been built, and there are almost as many types of codes as there are computers. Further, some peripheral devices have their own codes. Magnetic tape usually uses the same code as that employed in the computer proper, but because magnetic tape is used for transmitting at densities and speeds approaching the limit of engineering capability, these devices are prone to error. A parity bit is added for each character of information. Thus, we find **seven-track** and **nine-track** tapes used with characters represented by six-bit and eight-bit codes, respectively, with the addition of a parity bit.

Punched paper tape devices that employ five-, six-, seven-, or eight-bit codes are available. The codes are usually peculiar to these devices.

Most typewriter consoles use a printing head that looks much like a golf ball. The head can tilt and rotate to get the proper character into position to strike the paper. To tell this golf ball at what angle to tilt and what angle to rotate, a *Tilt/Rotate code* (T/R) has been developed. Characters transmitted to the type mechanism in EBCDIC must be converted to the T/R code to activate the mechanism properly. It is interesting to note that when the operator presses a key on such a typewriter, he produces a character coded in the paper-tape transmission code. This is normally converted into EBCDIC for transmission to the computer; it is also translated into the T/R code to energize the print ball. The operator can verify that both translations have occurred successfully, since the key struck produces only a code character; it prints the character he wants only if the code and two translations of the code are all correct.

REFERENCE

196 1 , Peterson, W. W. *Error Correcting Codes*. Cambridge, Mass.: M.I.T. Press.

I. FLORES

CODE, ERROR CORRECTING. *See*
ERROR CORRECTING CODE.

COGO. *See* **PROBLEM-ORIENTED LANGUAGES.**

COLLATING SEQUENCE

For articles on related subjects *see* **SORTING**; and **TABLE LOOKUP**.

For articles on related terms *see* **ASCII**; **EBCDIC**; and **KEY**.

The American National Standard *Vocabulary for Information Processing* defines collating sequence as:

An ordering assigned to a set of items, such that any two sets in that assigned order can be *collated*.

Collating, in the computer processing sense, derives from punched-card processing in which two decks of punched cards ordered in the same sequence on the same key are merged together (or collated), using a card collating machine. (The essence of the merging or collating methodology is explained elsewhere in this volume in the discussion of the merge **search** technique for table lookup).

Collating is often necessary in data processing applications; a good example is the collating of a set of updated records into a master file (or set) of records. This requires that both sets of records be ordered or sorted on a key in the same sequence (ascending or descending). This is illustrated by the following example of collating two sets into one, using a person's social security number as a key.

Set 1 (Updates)	
Key	Action
408-44-6083	Add
414-22-3598	Delete
414-36-1776	Add
Set 2 (Master)	
222-22-2222	
333-33-3333	
414-22-3598	
Set 3 (New Master after Collating and Updating)	
222-22-2222	
333-33-3333	
408-44-6083	
414-36-1 776	

In the example given, a social security number (a supposedly unique identity number which is often

COMIT

used in the United States for identifying each person) is the key, and these numbers are collated in ascending numerical sequence.

In a general sense, a collating **sequence** must be considered when assigning codes to the various characters to be represented in a computing system in order that collating may also be done in non-numerical keys. For example, consider the ordering or sequence for this set of characters:

A, X, 2, 7, a, b, ?, /, #

It must be determined in what order the various graphic characters of the set are to take. The preceding characters would order as follows, using two common character representation schemes.

ASCII (7 bit)		IBM EBCDIC (8 bit)	
Character	Code	Character	Code
#	010 0011	/	0110 0001
/	010 1111	?	0110 1111
2	011 0010	#	0111 1011
7	011 0111	a	1000 0001
?	011 1111	b	1000 0010
A	100 0001	A	1100 0001
X	101 1000	X	1110 0111
a	110 0001	2	1111 0010
b	110 0010	7	1111 0111

Of course **alphabetization** is achieved by ordering the binary codes representing the alphabetic characters. Also, codes representing the numbers should order properly, but the decision as to whether alphabetic characters should collate before or after numbers is somewhat arbitrary. The collating sequence for special characters is also an arbitrary choice, and various schemes are found in practice. ASCII (American Standard Code for Information Interchange) and EBCDIC (Extended Binary Coded Decimal Interchange Code) are perhaps the codes most often encountered, and the characters they represent will have a collating sequence corresponding to the value of the binary number code assigned to each character, as may be inferred from the tables in the articles on ASCII and EBCDIC.

When multiple characters are used to constitute a key for an item, the keys will collate in accordance with the composite character codes. For example, the name **JOHNSON** collates before **JONES** because the key representing **JOHNSON** has a lower value than the key for **JONES**. When EBCDIC codes are used, the keys for **JOHNSON** and **JONES** appear as follows (using a ten-character maximum length, left-justified key):

Name	Key (in Hexadecimal)
JOHNSON	D1 D6 C8 D5 E2 D6 D5 40 40 40
JONES	D1 D6 D5 C5 E2 40 40 404040

Note that hexadecimal **40** (binary 0100 0000) represents a blank-the character used to pad out the key.

C. E. PRICE

CO M IT. See **STRING PROCESSING LANGUAGES**

COMMAND AND JOB CONTROL LANGUAGES

For articles on related subjects **see CATALOG; JOB; INPUT-OUTPUT DEVICES; LANGUAGE PROCESSORS; LINKAGE EDITOR; TERMINALS; and TIME SHARING.**

For articles on related terms see **CENTRAL PROCESSING UNIT; MULTIPROGRAMMING; OBJECT PROGRAM; OPERATING SYSTEMS; and SOURCE PROGRAM.**

A command language (CL) or a job control language (JCL) is a language in which users of a data processing system (DPS) describe the requirement of their tasks (or job) to that system. Most data processing systems operate under the control of an operating system (OS). (Operating systems are also referred to as "monitors", "supervisors", and "command systems".) The **operating** system is the prime interface between a DPS and its users. The users interact with a DPS via the command or job control language of its operating system. The term "**command language**" is most often used when speaking of a time-sharing or interactive DPS, while "**job control language**" is used primarily in relation to batch processing systems. Here, we will use the term "**command language**" to mean both CL and JCL.

More specifically, users of data processing systems employ the command language to:

- 1. Identify themselves to the system for security and accounting purposes, and, in some instances, to inform the DPS which data files and file catalogs are to be used in processing their respective tasks.

2. Inform the DPS about the particular resources required by their tasks (e.g., amounts of primary and secondary storage, language translator(s) to be used, expected amount of central processor time for each task).

3. Specify input/output (I/O) devices required by their tasks (e.g., magnetic tapes, disks, line printer, plotter), and define the manner in which the information is or should be organized (or formatted) on these devices.

4. Specify what action the DPS should take in exceptional cases (e.g., errors in programs, missing or incorrect input data, I/O device malfunctions).

In batch systems, CL statements also separate the task(s) of each user from the tasks of other users which are in the same batch of tasks (also known as a "job stream") to be executed by the DPS.

Batch Command Languages. Early batch DPSs had no operating systems and were capable of executing only one task at a time. As a result, users of these systems controlled the execution of their respective tasks themselves; while the DPS was executing their tasks, such users often acted as operators of the DPS, and controlled the operation of the entire system. As data processing systems grew in complexity (and therefore in cost), this mode of operation became no longer economically feasible. Simple operating systems were developed to allow the DPS to sequence automatically the various user's tasks through the system. These early, simple batch operating systems executed one task at a time, either to completion or until some error made it impossible to continue a task [Rosin (1969); Jardine (1975); Barron (1971, chap. 9)]. In the latter case, the operating system would usually give the user (via a printed report) some rudimentary indication of what went wrong, and would *then* immediately proceed to the next user's task. The user had only a very limited ability to affect the behavior of the operating system. The system simply sequenced various user's tasks through the DPS, giving up on any task that did not behave exactly according to the user's (and the system's) expectations.

These batch operating systems utilized the DPS more efficiently, but at the price of increasing the overhead on the user's time: They forced the users to work in a much more formal and regulated fashion, sometimes with large delays (turnaround time) between the time a job was submitted by the user and the time when the user received the corresponding output*

Because of the large cost of DPSs, further attempts at making their use more efficient and at increasing their throughput resulted in the development of multiprogramming operating systems, which allow several independent tasks to use the DPS simultaneously. Thus, one task may be performing calculations while a second task may be reading a magnetic tape; a third, reading cards; etc. In addition, such concurrently executing tasks can, for instance, access the same disk unit (each task, of course, using only those portions of the disk which the operating system has assigned to that task) or specify that their output is to be printed on the same printer (in which case, the output of each task is saved by the operating system on some secondary storage device (e.g., a disk) and then printed when the printer becomes available). This mode of operation requires the users to inform the operating system about the specific resources which their tasks require.

This evolution of multiprogramming has had several notable effects:

1. Use of the DPS became more efficient.
2. Users were forced to state explicitly (and a priori) the resource requirements of their tasks. Users must state these requirements in a formal way, through the facilities of the command language, as opposed to remembering them, writing them on pieces of paper as "instructions to the operator," or coding them directly into their programs. Users can no longer assume that each user task has total control of the DPS.
3. It became possible for users to state their requirements in a more abstract fashion. Thus, one can say, for instance, that a task requires *three* tape drives, and the operating system chooses the actual tape drives to be used each time that this task is executed. This tends to minimize the interference between various user tasks, and allows a DPS to continue operating even when some of its resources (e.g., a tape drive) are unavailable because of failure or other reasons. In this fashion, a certain amount of independence from the actual physical configuration of the DPS is achieved.

Thus, with the passage of time, it became necessary for users to be able to state their requirements to the operating system in a more and more rigorous and detailed fashion. Simultaneously, the complexity of user tasks grew. Users want operating systems to take care of exceptional conditions (e.g., errors in input data) automatically without necessarily giving up on their tasks. To accomplish this,

COMMAND AND JOB CONTROL LANGUAGES

operating systems have to be able to make decisions based on what happens to a task while it is executing, and therefore command languages have to allow the user to state the rules and conditions for making these decisions.

As a result, the complexity of the user's interface (i.e., the CL) with the operating system has grown to accommodate these needs. As additional capabilities became needed in CLs, they were added, often in purely ad hoc ways, resulting in CLs that are very flexible and powerful, but also very complex, difficult to learn, unnatural to use, nonsystematic, and very often needlessly so [IBM (1972); Brown (1970); Barron and Jackson (1972)].

This increase in complexity has had several results:

1. The need, in most big data processing centers, for one or more (often full-time) "CL experts."
2. The development of procedure capabilities in CLs; these facilities allow a user to invoke, in a relatively simple fashion, a set of complex CL statements (i.e., a CL procedure) which that user, another user, or, more often, a "CL expert" has developed and has "debugged," and which is stored in the DPS under a specific name.
3. The emergence of research aimed at developing the theory and design of more general, systematic, simpler, and easier to use CLs [Dolotta and Irvine (1969); SHARE (1972); Gram and Hertweck (1975)].
4. The recent emergence of attempts at standardizing CLs. The purpose of such standardization is to make CLs less machine-dependent, just as was

done with several programming languages (e.g., Fortran, Cobol). In this context, see the concluding section below.

5. The increasing appeal to many users of time-sharing DPSS; this is due to the fact that, in addition to some other important factors, these systems very often tend to have CLs that are easier to learn and to use than the more traditional batch DPSS. We will return to this point below.

The part of the operating system which interprets the user's CL statements is often referred to as a "command language interpreter." The cards on which CL statements are punched are called "control cards." In order to give the reader a better understanding of a contemporary batch CL, we show in Fig. 1 a very simple job deck, the purpose of which is to compile, link edit, and execute (under the operating system known as OS/VS2 [IBM, 1972]), a Fortran source program.

The first card of the deck gives the name of the job (SAMPLE), the user's account 'number (1234) and name (JOHNDOE), and the priority class (K) which the user is requesting for this job. The second card indicates that the cataloged (prestored) CL procedure FORTCLG (for Fortran Compile, Link edit, and Go; i.e., execute) is to be executed. (Fig. 2 is a listing of FORTCLG.) The third card in Fig. 1 is a Data Definition card (DD), and it indicates that the Fortran source program is next in the deck. After the Fortran source program comes another DD card indicating that the data cards required by the program during the execution (GO) job step are next in the deck. After the data cards comes an end-

//SAMPLE JOB 1234,JOHNDOE,CLASS = K	00000010
/ / JOBDECK EXEC FORTCLG	00000020
//FORT.SYSIN DD *	00000030
...	
...The Fortran source program to be compiled goes here,	
...	
...	
//GO.SYSIN DD *	00001000
...	
...Data cards for the above Fortran program go here.	
...	
/*	00009000
//	00009999

Fig. 1. Example of a simple IBM OS/VS2 job deck.

of-data card (/*) and end-of-job card (/). In large decks, all the cards are usually numbered so that a deck can be put back into proper order should it be dropped or otherwise shuffled.

When the DPS (in this case, an IBM System/370 operating under OS/VS2) reads this deck, it verifies that the account number given on the first (JOB) card is a valid one, and then stores the job until a time when all the resources required for the first job step are available. The cataloged procedure `FORTCLG` (shown in Fig. 2) controls the execution of the job. We will not explain this procedure in detail. We do observe, however, that it consists of 20 cards (the first three cards and the last card are simply comments, which the operating system ignores), Procedure `FORTCLG` invokes two prestored programs: `IEYFORT` for the `FORT`, Fortran compilation, step; and `IEWLF440` for the `LKED`, link editing, step. For each job step, a number of additional files are specified by various, rather complex Data Definition (`DD`) statements, and various "default parameter s" are set. These parameters specify various choices which the user can override, such as the amount of secondary storage space allocated to a specific file. Three of the statements (those starting on cards 50, 100, and 140) have to be continued on

additional cards because of their length.

The first job step (`FORT`) compiles the Fortran source program (which follows card 30 in the deck shown in Fig. 1) into an object deck. If no errors are detected by the compiler, the second job step (`LKED`) link edits (combines) into a load module that object deck with other (already existing) programs required by that object deck. If this operation is successful, the load module is "loaded" (i.e., read) into the DPS main memory by the third job step (`GO`), and the user's program begins its execution, during which it presumably reads the data cards that follow card 1000 in the deck of Fig. 1.

Each job step produces output, which is stored on magnetic tape or disk. Some of that output is of a temporary nature and is discarded at the end of the job step. Other output (e.g., the object program) may be used by subsequent steps; still other output may be retained on disk or tape, as requested by the user, for subsequent use by other jobs. Finally, some output is usually returned, at a later time, to the user in the form of printouts and card decks.

Should any of the job steps run into some difficulty (e.g., errors in the source program), the printed output for that job step will so inform the user (occasionally in a rather cryptic fashion); and,

/*	-----	00000010
/*	FORTCLG -FORTRAN COMPILE, LINK, AND EXECUTE.	00000020
/*	-----	00000030
	PROC DECK = NODECK,SOURCE = ,MAP = NOMAP,LOAD = LOAD,LIST = NOLIST	00000040
//FORT	EXEC PGM = IEYFORT,REGION = 1 OOK,	00000050
	PARM = '&DECK,&SOURCE,&MAP,&LOAD,&LIST'	00000060
//SYSPRINT	DD SYSOUT = A	00000070
//SYSPUNCH	DD SYSOUT = B	00000080
//SYSLIN	DD UNIT = SYSDA,SPACE = (CYL,(1,1)),DISP = (,PASS)	00000090
//LKED	EXEC PGM = IEWLF440 ,COND = (4,LT,FORT),REGION = 96K,	00000100
	PARM = (XREF,LIST,LET)	00000110
//SYSLIB	DD DSN = &&FORTLIB1 ,DISP = (SHR,PASS)	00000120
	DD DSN = &&FORTLIB2,DISP = (SHR,PASS)	00000130
//SYSLMOD	DD DSN = &&GOSET(GO),DISP = (,PASS),UNIT = SYSDA,	00800140
	SPACE = (CYL,(1,1,1))	00000150
//SYSPRINT	DD SYSOUT = A	00000160
//SYSUT1	DD DSN = &&SYSUT1,UNIT = SYSSQ,SPACE = (1024,(100,50),,ROUND)	00000170
//SYSLIN	DD DSN = *.FORT.SYSLIN,DISP = (OLD,DELETE)	00000180
	DD DDNAME = SYSIN	00000190
//GO	EXEC PGM = *.LKED.SYSLMOD,COND = ((4,LT,FORT),(4,LT,LKED))	00000200
//FT05F001	DD DDNAME = SYSIN	00000210
//FT06F001	DD SYSOUT = A	00000220
//FT07F001	DD SYSOUT = B	00000230
/*	-----	00000240

Fig. 2. Example of a simple IBM OS/VS2 cataloged procedure.

COMMAND AND JOB CONTROL LANGUAGES

@RUN JDOE, 1234	00000010
@FOR,S	00000020
...	
... The Fortran source program to be compiled goes here.	
...	
...	
@XQT	00001000
...	
... Data cards for the above Fortran program go here.	
...	
@FIN	00009000

Fig. 3. Example of a simple UNIVAC EXEC-8 job deck.

at least in the example of Figs. 1 and 2, the job will be terminated at the end of that job step.

The reader should be warned that the OS/VS2 command language is probably the most complex CL in wide use today, and is very difficult to use. Barron and Jackson (1972) have said: "It is a language in which the articulate can speak powerful words of wisdom, but fluency is at the end of a long hard road." Many other batch CLs are significantly simpler, but also somewhat less flexible and versatile. The EXEC-8 CL, used on UNIVAC 1100 series of computers (UNIVAC, 1974), is an example of such a CL.

Fig. 3 shows an EXEC-8 job deck that is functionally quite analogous to the OS/VS2 deck of Fig. 1. The first (@RUN) card identifies the user (JDOE) and his account (1234). The second (@FOR) card requests the Fortran compilation of the immediately following Fortran source deck, and also requests a "short" (s) listing of that program. Card 1000 (@XQT) requests the execution of the just-compiled Fortran program, assuming that the compilation is error-free; the data for that execution follow the @XQT card. The @FIN card signals the end of the job.

Time-Sharing Command Languages.

Command languages meant to be used in a time-sharing or interactive mode are usually very much simpler than batch command languages. There are several reasons for this:

1. In a time-sharing mode the user most often types in a single CL statement at a time, observes the results of that statement, and then decides what to do next. Thus, the user does not have to decide and

explicitly state a priori what the system is to do under *all* possible conditions; he or she can make these decisions implicitly while interacting with the DPS.

2. The users interact directly and in real time with the DPS, as opposed to having to utilize the DPS through operators who submit their tasks.

3. Users of a time-sharing DPS are often geographically isolated from that DPS (e.g., by working at home). Under such conditions, simplicity and ease of use of the CL is a very important factor (Dolotta and Irvine, 1969).

4. Since CL statements are usually typed every time they are to be executed (as opposed to being punched on a cards that can be reused many times), it is vitally important that they be simple and short.

5. The DPS can guide the user by printing "prompting messages," thus making it less necessary for the user to remember all the details of the CL.

As a result, a great deal more attention has been paid to date to the human engineering aspects of time-sharing CLs than to that of batch CLs. In addition, time-sharing CLs and DPSs have a number of the better facilities found in batch systems. Thus, in many time-sharing CLs it is possible to construct and save cataloged command procedures for repeated use. In time-sharing DPSs, there is virtually no use made of card decks; all users' data files and programs are stored on line. Editing programs are usually provided in a DPS to allow users to conveniently create, examine, and modify their on-line files of programs and data.

We will again use an example to give the flavor of a modern time-sharing CL [Ritchie and Thompson (1974)]. Fig. 4 is a *verbatim* record of a short

```

login: janedoe
Password:
12/13/74 ■ System off the air after 1900 for preventive maintenance!
% date
Fri Dec 13 18:48:24 EST 1974
% ed quad.f
1752
/Cong/
54 format ('Conjugate roots; real = ',1pg16.6,' imag. = ',1pg16.6)
s/q/j/p
54 format ('Conjugate roots; real = ',1 pg16.6,' imag. = ',1 pg16.6)

W
1752
q
% fc quad.f
% mv a.out quad
% quad

Please enter the values of a, b, and c:
1 -2 1
a = 1 .00000 b = -2.00000 c = 1.00000
Double real root; root = 1.00000

Please enter the values of a, b, and c:
1e4 2e3 3e2
a = 10000.0 b = 2000.00 c = 300.000
Conjugate roots; real = 0.100000 imag. = 0.141421

Please enter the values of a, b, and c:
u -6.25
a = 0.00000 b = 0.00000 c = -6.25000
== > Coefficients imply that -6.25000 = 0 (!)

Please enter the values of a, b, and c:
1,, -6.25
a = 1.00000 b = 0.00000 c = -6.25000
Two real roots; first = 2.50000 second = -2.50000

Please enter the values of a, b, and c:
1.667e-3 6.375e+5
a = 1.667000e-03 b = 637500. c = 0.00000
Two real roots; first = 0 second = -3.824236e+08

Please enter the values of a, b, and c:
-
a = 0.00000 b = 0.00000 c = 0.00000
All coefficients are zero, Program terminated.

% date
Fri Dec 13 18:51:07 EST 1974
%

```

Fig. 4. A short terminal session with a time-sharing DPS.

COMMAND AND JOB CONTROL LANGUAGES

```
c quad.f - interactive Fortran program to solve quadratic equations of
c the form:  $a*x**2 + b*x + c = 0$ ; the following cases are considered:
c   if  $a \neq 0$  &  $c \neq 0$       = = > general case      (500);
c   if  $a \neq 0$  &  $c = 0$       = = > roots are 0 and  $-b/a$  (400);
c   if  $a = 0$  &  $b \neq 0$       = = > only root is  $-c/b$     (3013);
c   if  $a = 0$  &  $b = 0$  &  $c \neq 0$  = = > input error      (200);
c   if  $a = 0$  &  $b = 0$  &  $c = 0$  = = > terminate program.
100  write (6,10)
      10  format (/, 'Please enter the values of a, b, and c:')
      read (5,12) a, b, c
      12  format (3916.6)
      write (6,14) a, b, c
      14  format ('a = ',1pg16.6,' b = ',1pg16.6,' c = ',1pg16.6)
      if (a .ne. 0.0 .and. c .ne. 0.0) goto 500
      if (a .ne. 0.0) goto 400
      if (b .ne. 0.0) goto 300
      if (c .ne. 0.0) goto 200
      write (6,18)
      18  format ('All coefficients are zero. Program terminated./')
      stop
200  write (6,20) c
      20  format (' = = > Coefficients imply that ',1pg16.6,' = 0 (!)')
      goto 100
300  x = -c/b
      write (6,30) x
      30  format ('Single real root; root = ',1pg16.6)
      goto 100
400  x = -b/a
      write (6,40) x
      format ('Two real roots; first = 0 second = ',1pg16.6)
      goto 100
c      General case (a and c are both nonzero)
500  x = b/(2.0*a)
      disc = b**2 - 4.0*a*c
      sdisc = sqrt(abs(disc))/(2.0*a)
      if (disc .gt. 0.0) goto 560
      if (disc .lt. 0.0) goto 540
      write (6,52) x
      52  format ('Double real root; root = ',1 pg16.6)
      goto 100
540  write (6,54) x, sdisc
      54  format ('Conjugate roots; real = ',1 pg16.6,' imag. = ',1 pg16.6)
      goto 100
560  x1 = x + sdisc
      x2 = x - sdisc
      write (6,56) x1, x2
      56  format ('Two real roots; first = ',1pg16.6,' second = ',1 pg16.6)
      goto 100
      end
```

Fig. 5. A simple Fortran source program (adapted from Kernighan and Plauger, 1974).

“terminal session” with such a system; *except that*, for ease of understanding, we have underlined everything that was typed by the user. The purpose of this session is, again, to compile and execute an already existing **Fortran** program.

As soon as the user has dialed-up the DPS from a terminal, the system asks for a “login” code, which the user types (*janedoe*). The system then asks for the user’s secret password to make sure that the user is indeed the person who is authorized to log into the system with the code *janedoe*; at this point, the DPS also turns off the printing mechanism of the terminal, thus preserving the secrecy of the password by making it invisible. Once the user has entered the correct password, the printing is turned back on, and the user is informed, via a “message of the day,” that the system will be unavailable after 7 P.M. The percent sign (%) is a system-prompting message, or “prompt,” for the next command. The user asks for the date (knowing that this will cause the system to print both the date and the time). Observing that there is still in excess of 11 minutes before the system is to shut down, the user continues with the substance of the session.

The next command indicates that the user wishes to edit (*ed*) a **Fortran** source program that is stored in an on-line file called *quad.f*, the contents of which are shown in Fig. 5. The system reads this file and acknowledges the request by printing the number of characters in that file (1752). (Note that, unlike most batch systems, most interactive systems accept both upper and lower-case input; this is possible because most interactive terminals (unlike card punches) can type in both upper and lower case.) The user asks the editing program (the “editor”) to find a source statement that contains the letters *Cong*; the system prints that line; the user corrects the typographical mistake by substituting (*s*), the first occurrence of the letter g by the letter j and printing (*p*) the corrected line. Satisfied with the result, the user writes out (*w*) the modified program onto on-line storage; the system reports that the length of the file is still 1752 characters. The user then quits (*q*) using the editor; the system prompts for the next command via the percent sign. The user asks that the just-modified file be compiled by the **Fortran** compiler (*fc*); the compiler, having found no errors, causes the system to make a load module from the resulting object program (leaving that load module in a file called, by convention, *a.out*), and then to prompt for the next command. The user moves (*mv*) the load module *a.out* to a file called *quad* (so that it will not be overwritten next time she or he uses the compiler); and then, simply by typing

the name of that module (*quad*), causes it to be executed.

At this point, the compiled **Fortran** program begins interacting with the user. The purpose of this program is to solve quadratic equations of the form $ax^2 + bx + c = 0$, given the values of the coefficients *a*, *b*, and *c*. The program prompts the user for the first set of coefficients, which the user types in. The program “echoes” the values of the coefficients (*a = 1.00000 , . . .*), prints the type of the solution and the value of the root (1.00000), and prompts for the next set of coefficients. The user solves three more equations, making a mistake on the second one of these, and thus having to do it over. Note that if the user does not supply the values of one or more coefficients, then these coefficients are automatically set to zero. Furthermore, the **Fortran** program assumes that it is to terminate itself if all three coefficients are zero (see the three lines immediately above statement 200 in Fig. 5). Therefore, the user terminates the program by simply typing an empty (blank) line. At this point, the system prompts for the next command. The user asks again for the time, discovers that the session lasted a bit under 3 minutes, and instead of entering another command, turns off the terminal, thus disconnecting it from the DPS.

General Observations. Unlike programming languages (e.g., **Fortran**), command languages available for various computer vendors’ DPSs have very little in common with each other; for example, as can be deduced from Figs. 1 and 3, there is no compatibility between IBM OS/VS2 CL and UNIVAC EXEC-8 CL. In fact, even the terminology used to describe the various CL facilities is different between the two. Therefore, while it is relatively easy to convert a **Fortran** program from one of these systems to the other, the conversion of the corresponding CL statements is very difficult. This situation leads to a great deal of inefficiency, and is a very unfortunate one.

It is becoming more and more common to provide both batch and time-sharing services on the same DPS. Users of some of these systems [IBM (1970), IBM (1972)] often are faced with the need to learn two CLs if they wish to use the system in both modes. Furthermore, since the two CLs in such a system must coexist (e.g., be able to access the same files), both tend to be less than optimal for their respective tasks. For historical reasons, in such a situation the time-sharing CL usually “lives under” the batch CL and must adjust to it, acquiring in the process many of the undesirable characteristics of its

COMMUNICATION CONTROL UNIT

"parent" (IBM, 1970).

On the other hand, because the state-of-the-art in the area of CL design and implementation is still relatively rudimentary, it is not clear that attempts at standardizing CLs (OSCLTG, 1975) are desirable at this time. It is conceivable that such premature standardization of CLs might in fact slow down progress in this area.

Nonetheless, it is likely that CLs in the future will be designed in more systematic ways and with more attention being paid to human engineering factors than has been the case in the past.

REFERENCES

1969. Dolotta, T. A., and C. A. Irvine, "Proposal for a Time-Sharing Command Structure," *Information Processing* Vol. 68, pp. 493-498. Amsterdam: North Holland Publishing.
1969. Rosin, R. F. "Supervisory and Monitor Systems," *Computing Surveys*, Vol. 1, No. 1, pp. 37-54.
1970. Brown, G. D. *System/360 Job Control Language*. New York: John Wiley.
1970. Data Processing Division, IBM Corp. *IBM System/360 Operating System: Time-Sharing Option Command Language Reference*. Form GC28-6732. White Plains, N.Y.: IBM.
1971. Barron, D. W. *Computer Operating Systems*. London: Chapman and Hall; New York: Barnes and Noble.
1972. Barron, D. W., and I. R. Jackson. "The Evolution of Job Control Languages," *Software-Practice & Experience*, Vol. 2, No. 2, pp. 143-164.
1972. Data Processing Division, IBM Corp. *OS/VS JCL Reference*. Form GC28-0618. White Plains, N.Y.: IBM.
1972. SHARE Inc. "Command Language Position Paper." SSD No. 221, Serial No. C-5647, April 15, 1972; also SSD No. 226, Serial No. C-5715, Sept. 25, 1972. Chicago, Ill.
1974. Kernighan, B. W., and P. J. Plauger. *The Elements of Programming Style*. New York: McGraw-Hill.
1974. Ritchie, D. M., and K. Thompson. "The UNIX Time-Sharing System," *Communications of the ACM*, Vol. 17, No. 1, pp. 365-375.
1974. UNIVAC DIVISION, Sperry Rand Corp. *UNIVAC 1100 Series Operating System Programmer Reference*. Form UP-4144, Rev. 3. Blue Bell, Pa.
1975. Gram, C., and F. R. Hertweck. "Command Languages : Design Considerations and Basic Concepts," in C. Unger (Ed.), *Command Languages*. Amsterdam : North Holland; New York: American Elsevier, pp. 43-69.
1975. Jardine, D. A. "The Structure of Operating System Control Languages," in C. Ungér (Ed.), in *Command Languages*. Amsterdam: North Holland; New York: American Elsevier, pp. 27-42.
1975. OSCLTG, "The Operating Systems Command Language Task Group Technical Report," in C. Unger (Ed.), *Command Languages*, Amsterdam: North Holland; New York: American Elsevier, pp. 353-388.

T. A. DOLOTTA

COMMUNICATION CONTROL UNIT

For articles on related subjects see **CHANNEL**; **DATA COMMUNICATIONS**; **DATA COMMUNICATION NETWORKS**; **FRONT END**; **MULTIPLEXING**; **TELEPROCESSING SYSTEMS**; and **TERMINALS**.

Modern large-scale computers generally have the capability of **accepting** data or jobs originating from remote terminals or computers. This necessitates some form of a data communication network to transmit the data. Such a network consists of a set of nodes connected by a set of links as shown in Fig. 1. The nodes may be the host computer(s), terminals or some type of communication control units, while the links are the communication channels, which are usually private or switched lines leased from a common carrier. Because transmission over these lines is generally analog, while the signals at the nodes must be digital, data sets are used to provide the interface between node and link.

The name "**Communication Control Unit**" is vague. It may include such units as message switchers, remote terminal controllers, concentrators, front-end communication controllers, or simple multiplexers. The latter are usually hardwired units, while the first four may be hardwired or programmable, in which case they are computers programmed to perform various **communication-oriented** tasks. Such computers are referred to as "communication processors." The main function of these controllers is to increase the efficiency and decrease the cost of the total network.

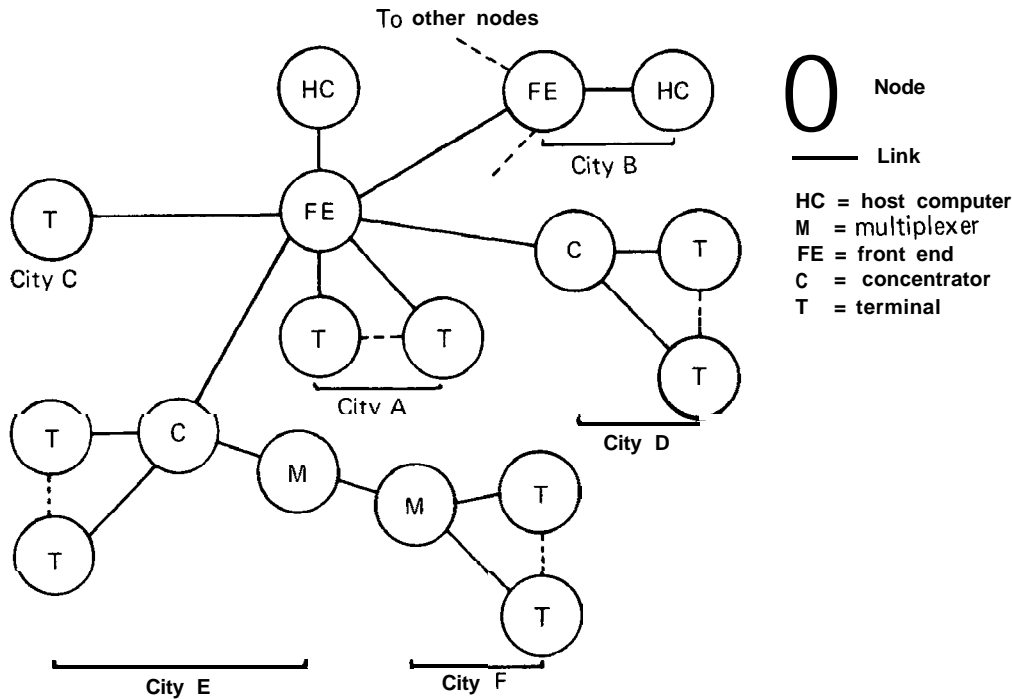


Fig. 1. Example of a data communication network.

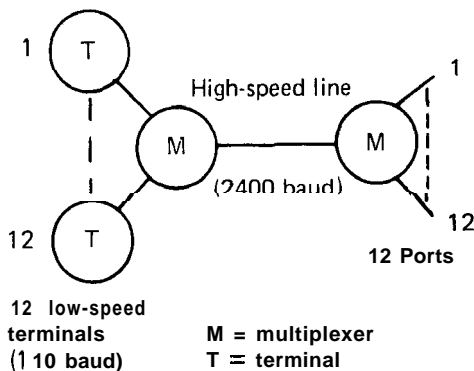


Fig. 2. Example of the use of multiplexer.

Multiplexers. Multiplexing permits the transmission of several lower-bandwidth data streams over a single higher-bandwidth line, as shown in Fig. 2. There are two basic techniques: frequency division multiplexing (FDM) and time division multiplexing (TDM). FDM divides the frequency spectrum of a line into several smaller frequency bands. Data from terminals is sent over these smaller bands by frequency-shift keying. At the receiving end, the various frequency bands will be reconverted into their original data by means of bandpass filters.

TDM is very similar to the action of a commutator. Each terminal is sampled one by one for one "bit time" in round-robin fashion, and the samples are assembled into a serial data stream. At the receiving end the serial stream is disassembled and the data is routed to the correct terminal or computer port.

Multiplexers are usually simple hardwired units. Their main use is in reducing line costs through reduction of the number of lines needed to handle remote terminals.

Concentrators. The term "concentrator" is usually reserved for a small computer programmed to perform the task of a multiplexer. In practice, terminals do not transmit or receive data at maximum bit rates over sustained periods of time. Since the FDMs and TDMs have no buffering, they must be capable of handling data under worst-case conditions, that is, at maximum bit rates. Because of its stored program flexibility and buffering capability, a concentrator can pack the higher-speed line to its maximum capacity. This allows the concentrator to multiplex more low-speed lines onto a given line having a higher speed than an FDM or TDM system. The buffering absorbs the peak loads, while occasionally sustained peak loads can be accommodated by software. This may be done by tem-

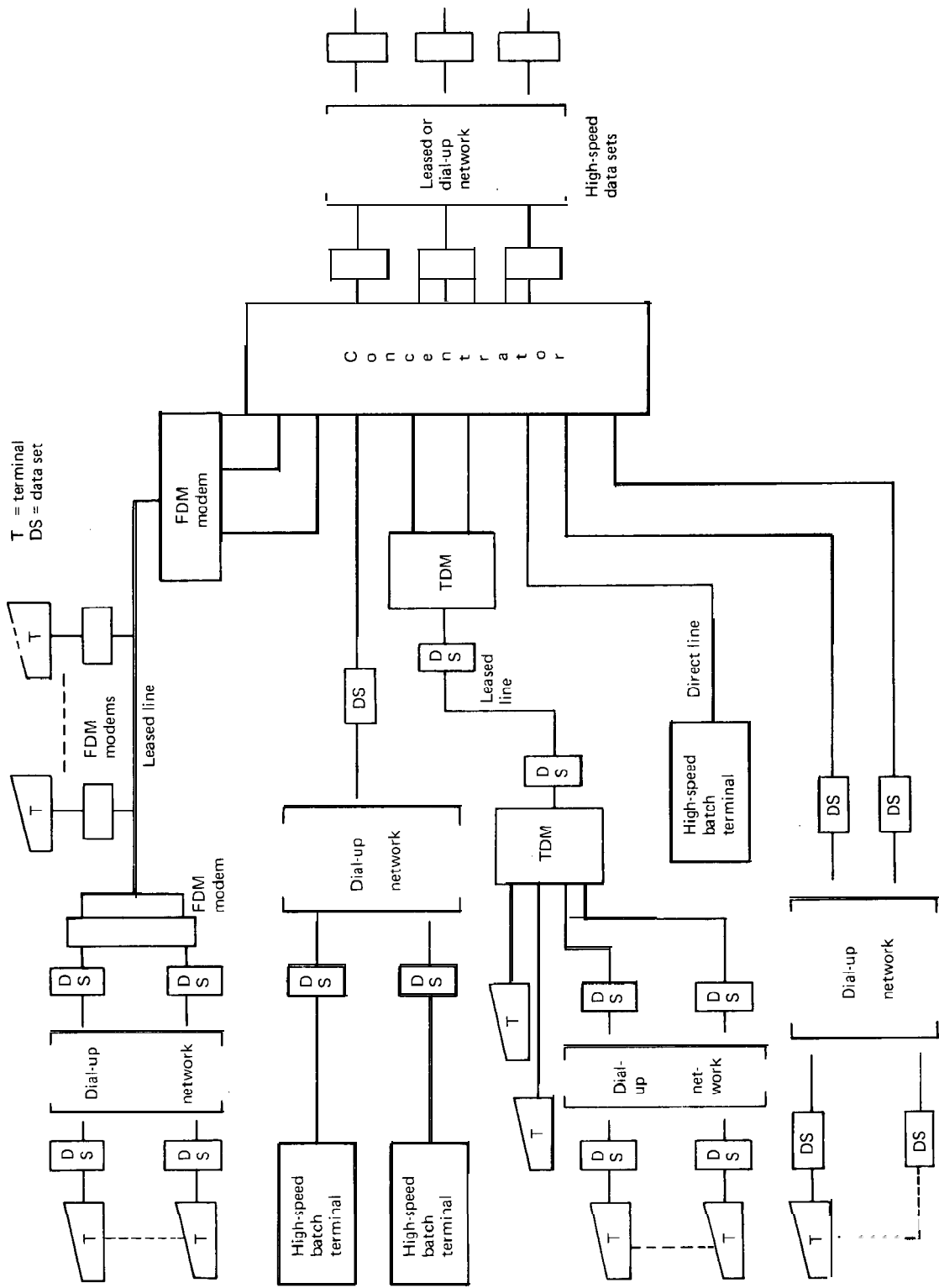


Fig. 3. A concentrator supporting a large variety of local and remote terminals.

COMMUNICATION CONTROL UNIT

porarily inhibiting transmission to some terminals or sending a command to a terminal to reduce its transmission.

Even though a concentrator is more expensive than a simple multiplexer, it can be far more efficient. Moreover, it can be programmed to perform additional tasks such as control of local terminals, code conversion, line polling, error detection, and other control functions at no additional hardware cost. Furthermore, interfaces may be built to enable it to accept data from a far wider range of local or remote equipment than could a conventional hardwired multiplexer.

Fig. 3 shows a concentrator accepting data from a number of terminals of various types and concentrating it onto three high-speed lines. The terminals are connected to the concentrator by dial-up, leased or direct lines as in the case of the local high-speed batch terminal. Notice the flexibility afforded by the TDM and the FDM. The former supports local and dial-up terminals at the remote end. The FDM modem may also support local or dial-up terminals. Furthermore, the leased line may pass through several cities, each of which may have a "drop" to a terminal. Such use of leased lines is known as "multidropping."

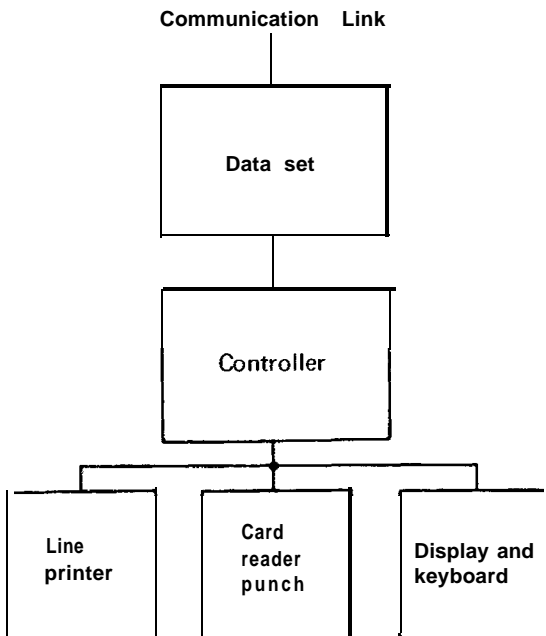


Fig. 4. A simple remote batch terminal and controller .

Remote Terminal Controllers. Some types of remote terminals require sophisticated controllers. An example is a bank of several CRT terminals or a remote batch-processing station consisting of a line printer and card reader/punch, as shown in Fig. 4. Again, such controllers may be hardwired or programmable. The latter are becoming more attractive because of their flexibility. Thus, they may be programmed to emulate IBM, CDC, or UNIVAC batch-processing stations merely by a program change. Their use as "intelligent terminals" is also becoming popular.

Front-End Communication Controllers. These provide the interface between the main computer and the communication network, as shown in Fig. 5. In the past these were hardwired, and consequently the mechanism of data transfer for the various terminals, error control, and data set control required a large amount of host computer resources in the form of CPU time and core space. These hardwired controllers are of ten called "transmission control units." A programmable unit is usually referred to as a front end processor or simply as the front end. Programmable front ends are becoming more and more popular because they are usually lower in cost and provide far better performance. Many of the functions (such as code conversion, message assembly, and editing) that were previously handled by the host computer may now be pro-

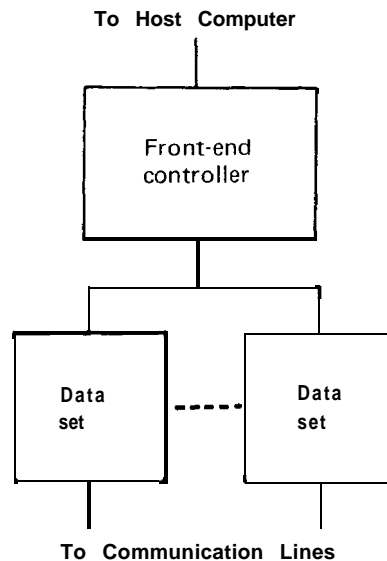


Fig. 5. A front-end communication controller or transmission control unit.

COMMUNICATION CONTROL UNIT

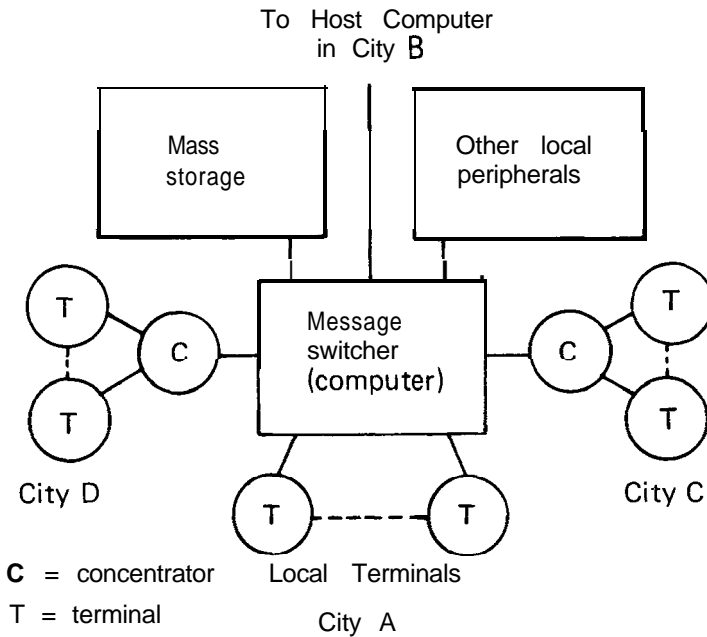


Fig. 6 A simple message-switching system.

grammed and handled by the front end. Overhead on the main system is thus reduced.

Message Switchers. A message switching system essentially accepts data from a large number of terminals, stores it, and then forwards it to other terminals as required. For this reason it is sometimes called a "store and forward" system.

A message switching system must have all the input-output and line control capabilities of a concentrator. In addition, it must have some mass storage in the form of disk and/or magnetic tape for the various messages, as shown in Fig. 6. Otherwise, the essential difference between it and the concentrator is to be found in the software because of the different functions each system must perform.

An interesting type of message switcher is the IMP (Interface Message Processor) used on the ARPA network (Hart et al., 1970).

Communication Control Unit Hardware. The essential components of programmable communication control units are (see Fig. 7):

1. One or more processors.
2. Flexible input-output channels.
3. Line interface units.
4. An interface to the host computer in the case of front ends.

The processor is a small stored program computer with data channels. The memory should be large enough to store the required program and provide adequate buffering for all lines. The instruction execution and memory cycle times should be small to allow many lines to be serviced without overruns, i.e., without loss of data. The instruction set should be oriented toward use in a communication environment where the main aim is the movement and manipulation of data. A powerful set of logical, bit manipulative, character-moving, list-processing, and interrupt-handling instructions is necessary. Computers with dynamic control storage are desirable to enable "tailored instructions" to be written in microcode. Such instructions can improve the throughput significantly.

Flexible communication processors must be easily interfaced to a large number of various terminals and data sets. The data rates on these may vary; they may be buffered or unbuffered and work in synchronous or asynchronous modes. To accommodate all of these interfaces, the input-output structure must be very flexible. For high-speed lines (above 40,000 baud, say), special channels with direct access to memory are desirable. Such channels can access memory on a cycle-stealing basis and provide no interference to the processor once a transfer is initiated. For the low- and medium-speed line, a time division multiplexer channel with

COMMUNICATION CONTROL UNIT

maskable multilevel interrupts and short interrupt response times is desirable. The address and status of the interrupt-causing device should be available quickly, and branching to the routine servicing the interrupt should be rapid. This may be accomplished by automatic swapping of current and new "program status words" that reflect the location of the instruction to be executed, the condition code, and the state of the interrupt masks. Several such program status words should be provided, one for each type of interrupt. The handshaking on the high- and low-speed bus should be as simple as possible to ease the design of the various interfaces to terminals and data sets (modems).

The interface units link the channels with the terminals or data sets terminating the communication lines. A general-purpose interface should be speed-independent and handle synchronous or asynchronous transmission. Because of the wide variety of speeds, this is not always possible. To simplify hardware, two or three different types of interface units are usually built, each optimized for a given speed range. Control of data sets and terminals is

done by hardware or software, usually the latter. The hardware inputs data-set status and outputs control signals. The software senses this status, interprets it, and outputs appropriate control signals. The processing is small but the flexibility is high, since any changes in equipment may be accommodated by appropriate changes in software. This makes the interface equipment independent and prolongs the usability of the system.

Since transmission between nodes is usually serial by bit, while on the bus it is parallel, the interface must perform the necessary conversion (character assembly and disassembly). This also may be done by software or hardware. Assembly by software results in a very cheap interface, since the characters are actually assembled in core, but the software overhead is high. This overhead is greatly reduced when hardware conversion or sampling is used. When the hardware has assembled or disassembled a character, an interrupt is generated and the computer fetches this character or sends the next one for disassembly. This technique is always used for high-speed lines and, considering the current

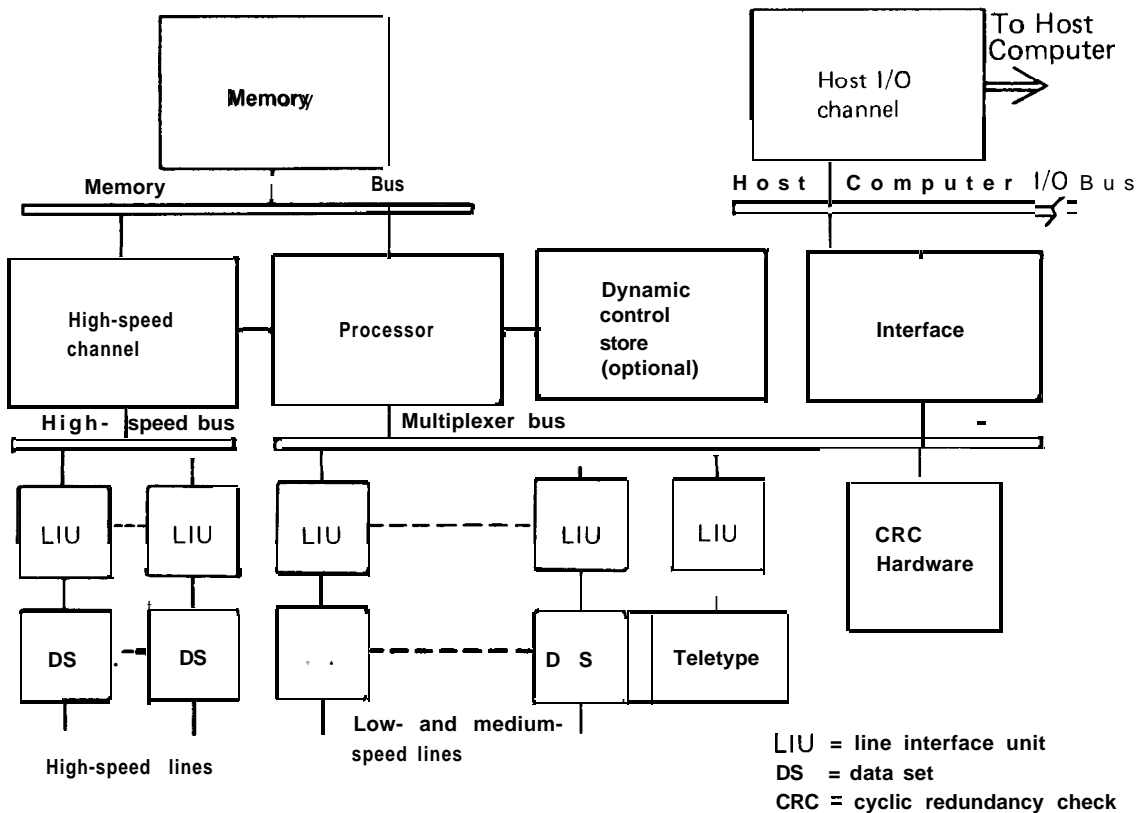


Fig. 7. Block diagram of a programmable front end.

COMMUNICATION CONTROL UNIT

prices of circuits, it is economical even for a large number of low-speed lines.

An interface to the host computer is required for front ends. A multiple address or a single address interface may be used. In a multiple address interface, each address corresponds to a terminal. In a single address interface, the address corresponds to the front end. In the single address case, as characters are passed between the front end and the host, they must be accompanied by an "address field" specifying the terminal to or from which they are coming. With a modest number of terminals, multiple address interfaces are more common. As the number of terminals becomes very large, the single address interface will become more popular.

A simple, programmable front end is shown in the block diagram of Fig. 7. Other types of programmable controllers would be similar, but they would not require the interface to the host. Note the Cyclic Redundancy Check (CRC) hardware in Fig. 7. Error control is very important in these systems; consequently, some form of error detection is usually included. If a cyclic code is used, evaluating the CRC characters by hardware is very common to avoid large software overhead.

It is difficult to generalize the structure of hardwired communication controllers. Basically, they consist of line interface units, character assembly and disassembly registers, some buffers (especially for high-speed synchronous lines), and a control unit. The complexity of the latter depends upon what functions are performed by software on the host and what functions are left to be performed by the hardware. The limitations of these controllers therefore become quite apparent. A sophisticated controller may require a disproportionate amount of hardware, and any changes in operation or addition of new types of equipment may result in insurmountable problems.

Hardwired Versus Programmable Controllers. For small, unsophisticated systems serving few terminals, hardwired controllers have a distinct advantage over their programmable counterparts. Examples are simple FDM or TDM systems, or simple terminal controllers. For sophisticated systems with a large number of terminals, programmable controllers can be far more efficient on a cost-performance basis. They may be programmed to perform a wide range of functions, some of which are not even available on the most expensive hardwired controllers. Some of these functions include:

1. Line polling and control.
2. Adaptive line-speed control.
3. Code conversion.
4. Message assembly and editing.
5. Error control.
6. Data compression.
7. Simple syntax checking.
8. Automatic loading of network programs from the host.
9. Line monitoring.
10. Buffering and concentrating.
11. Message recording.
12. Message answering and routing.

These functions, and many more, relieve the host, allowing it more time for data processing. It should be remembered, however, that as more of these functions are performed by the controller, the fewer lines it can handle.

Besides relief of the host, there are other indirect advantages to programmable controllers. One of these is terminal independency. A terminal not supported by the host may be made to appear like another terminal that is supported by having appropriate software routines. The monitoring and auto-loading functions can improve the reliability and maintainability of the network. Bad lines and bottlenecks may be quickly spotted by monitoring all activity, while autoloading (automatic loading of the controller with various programs from the host) may enable the identification of faults and suggest corrective action by running a series of diagnostics over the entire network. The ease with which more terminals or new terminals may be added increases the effective life of these controllers and the network



Fig. 8. IBM 3704 transmission control units.

COMMUNICATIONS AND COMPUTERS

as a whole. This, together with the relief provided to the host, also enables the latter to have a longer useful life.

Although the above advantages make programmable controllers very attractive, they have not been around long enough for their capabilities to be fully assessed and exploited. One reason for this is that the operating system on the host has been oriented, in the past, toward hardwired controllers. This will change rapidly in the years to come.

Future Trends and Conclusions. The advent of independent data channels and multi-programming enabled computers to support simultaneously many remote terminals. This resulted in the development of data communication networks, requiring several different types of communication control units. In the past these tended to be hardwired. As the networks grew, these controllers became very costly and imposed large overhead on the host. This, together with decreasing costs of small computers, led to the development of programmable controllers, which are more flexible and can be programmed to perform a wide variety of functions at lower cost. Thus, a single programmable controller could simultaneously be a terminal controller for some local terminals and a concentrator or message switcher for some of the remotes. Such flexibility will result (in some cases it already has) in announcements of new programmable controllers neatly integrated with the network and the host computer. The new teleprocessing systems will reflect this change. They will be simpler and will greatly improve the efficiency of the host.

REFERENCES

1970. Heart, F. E., et al. "The Interface Message Processor for the ARPA Computer Network," *A FIPS Conference Proceedings*, Spring Joint Computer Conference, pp. 55 1-567.
1971. Sobolewski, J. S. "Programmable Communication Processors," *First International Conference on Computer Communication*, Washington, D.C., October, pp. 380-389.
1972. Ball, C. J. "Communications and the Mini-computer," *Computer*, September, pp. 13-21.

J. S. SOBOLEWSKI

COMMUNICATIONS. See DATA COMMUNICATION NETWORKS; and DATA COMMUNICATIONS.

For articles on related subjects see **ARPA NETWORK; COMMUNICATION CONTROL UNIT; COMPUTER NETWORKS; DATA COMMUNICATIONS; and PACKET SWITCHING.**

For articles on related terms see **BLOCK DIAGRAM; and STORED PROGRAM CONCEPT.**

The first truly automatic digital computers were a direct outgrowth of the telecommunications switching art, using electromechanical components common to telephone central offices. Today the situation is reversed, with the computer imposing the new requirements and technology on telecommunications, and on switches in particular. In this article we consider the utilization of computers within communications networks.

Computers are predominantly used in communications networks as switches, or to control switches. In the first case, which is restricted to data communications, the computers are the switching nodes of the network. The interconnections and line capacities are determined by the geographical distributions of the message traffic. Each switching computer accepts messages from data terminals or other computers and records it in transient memory, where it is held until a communications channel is available to the computer serving the message destination. Multiple addressing is possible for any message. This type of switching is known as "store and forward," or "message," switching. In the second case, the computer controls equipment (commonly called the "matrix") that makes a metallic connection between origin and destination. The subscriber then uses the end-to-end communication channel capacity to transmit his information, either voice or data. (Today, voice traffic predominates.) There are also computers used as communication front ends, concentrators, etc., but these are auxiliary to the dominant roles above.

Computers in Message Switching.

Fig. 1 shows the main functional elements of a typical message switch. Incoming messages are buffered in the incoming line buffers. These buffers need provide only relatively limited storage to accommodate small delays in the availability of the main processor, which must share its capacity with many other assigned tasks. The message is placed in memory, and the address portion of the message (also known

COMMUNICATIONS AND COMPUTERS

as the "leader") is extracted. Depending upon the sophistication of the system, the address may be in a language that closely resembles English for the convenience of the user or close to machine language for more automated data systems. In any case, the computer can provide many service features such as inserting multiple addresses from an abbreviated code word and a standard list, giving priorities to authorized messages, permanent storage for record, and code conversion for different terminals where one terminal might use an older 5-bit per character code and the other a modern 8-bit per character code. Upon completion of the required processing the messages are outputted to the selected trunk via a small output buffer. Obviously not all messages will require the same processing time, and the size of the buffers and the message handling capacity are highly dependent upon processing rate. To design a switch properly requires knowledge of the theory of stochastic processes and "birth-death" processes in particular (Feller, 1967).

The analysis of switch performance must proceed from basic assumptions about the statistics of the messages the switch is being asked to process. The most commonly made assumptions are: The messages arrive at random and their distribution function is Poisson, with some average arrival rate λ ; the switch completion rate is also random, with an exponential distribution function and an average completion rate μ . Using these assumptions, which many years of telephone traffic engineering have shown to be remarkably accurate, the following

formulas can be derived (Hillier and Lieberman, 1967):

1. Expected input queue plus traffic in process length, L .
2. Expected input queue length, L_q .
3. Expected waiting time in system, W .
4. Expected waiting time in input queue, W_q .

$$L = \frac{\lambda}{\mu - \lambda} \quad (1)$$

$$L_q = \frac{\lambda^2}{\mu(\mu - \lambda)} \quad (2)$$

$$W = \frac{1}{\mu - \lambda} \quad (3)$$

$$W_q = \frac{\lambda}{\mu(\mu - \lambda)} \quad (4)$$

Much of the behavior of the computer in message switching applications can be explained in terms of Eqs. 1-4. Note that as the message-arrival rates (or requests for service) approach the completion rates, the queue lengths and waiting times approach infinity. In actual service the system will saturate and messages will be lost or denied access to the system. For a given traffic capability the hardware and software must be capable of supporting a

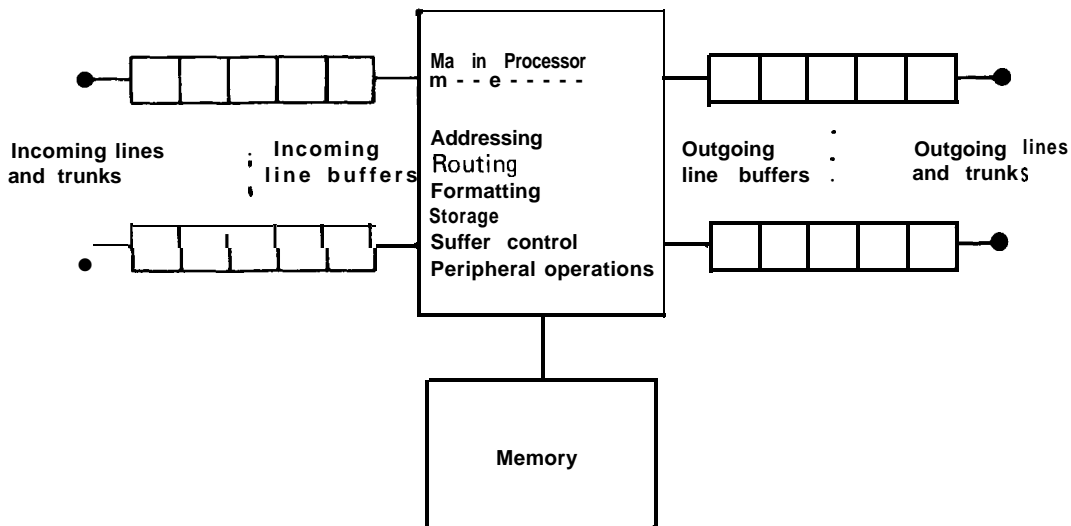


FIG. 1. Functional block diagram of typical message switch.

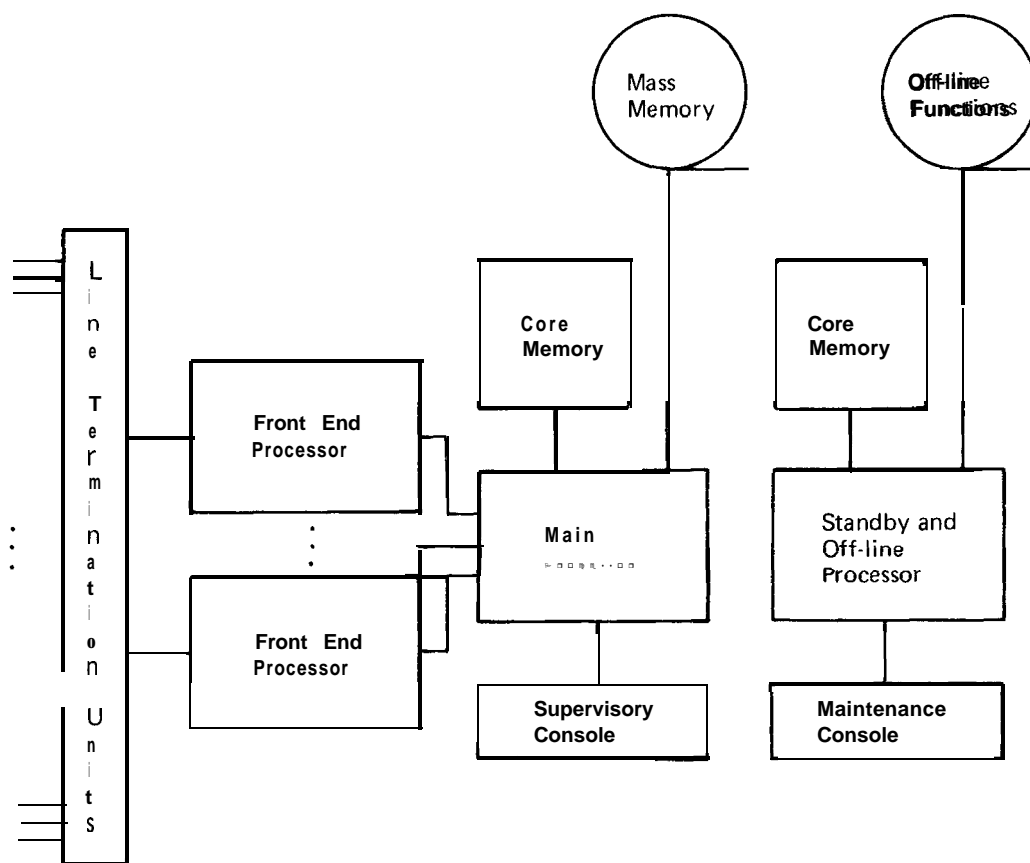


Fig. 2. Hardware functional block diagram of large message switch.

completion rate of about twice the expected arrival rate to provide adequate performance margins. Higher completion rates increase performance by increasingly smaller marginal improvements. Any rate lower than about 1.25 times the arrival rate is subject to overload by small fluctuations. In communications applications where the computer is an essential element in providing network service, practically all such applications are queueing systems of one form or another, and Eqs. 1-4 will at the least estimate their expected performance.

Fig. 2 shows a simplified hardware block diagram of a typical large-scale message switch. It should be noted that the complete switch is usually made up of more than one computer, and in fact is a true multiprocessor. In addition to the minicomputers performing line buffering, there are duplicate main processors, one being used as standby and to provide peripheral off-line computation. This is a common practice in switches of this size. In such a system, speed of service will vary, depending upon

the distribution and type of traffic and the topology of the network.

Fig. 3 shows some of the possible internodal connections in common use. Figure 3(a) is a star configuration with the message switch (CP) at the center and user terminals (C, concentrator; T, terminal) connected to it. This configuration is typically used with a large computer complex at the node and for messages of reasonably large size. However, large numbers of short messages can badly overload the computer. But this is precisely the type of traffic generated by users accessing computers in an interactive, time-sharing mode and by much computer to computer traffic. Furthermore, fast network response times of the order of 0.5 sec are imperative. Fig. 3(b) is the general topology of the ARPANET developed by the Advanced Research Project Agency of the Department of Defense to handle such traffic. As such, it represents the largest and most advanced use of computers in communications today.

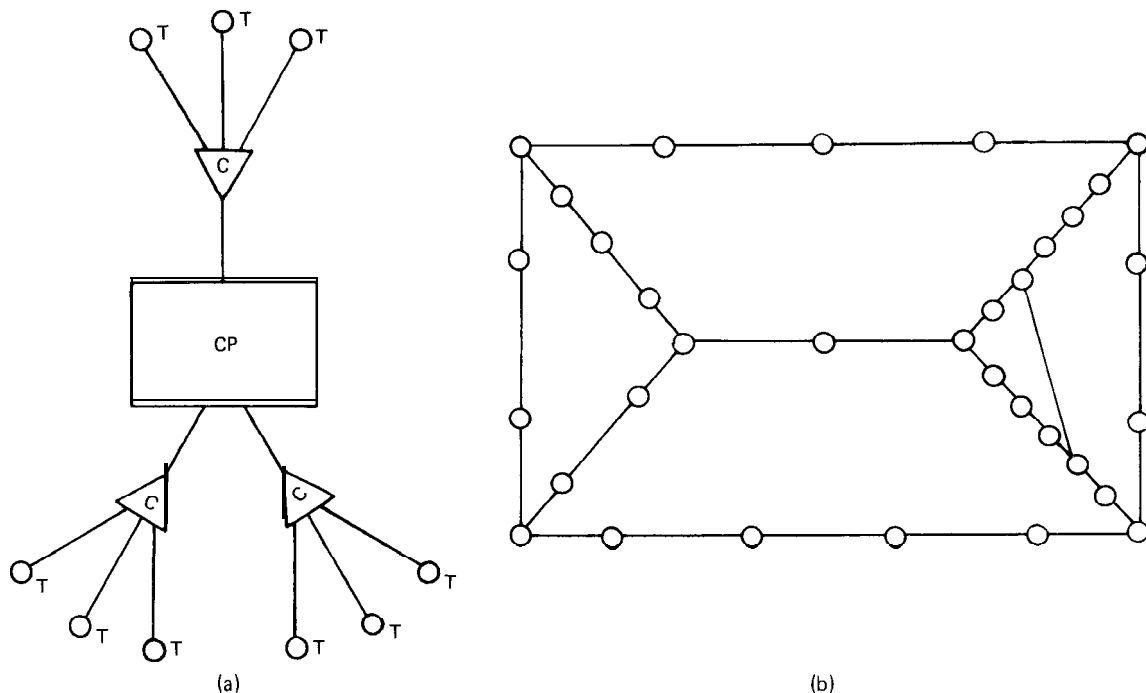


Fig. 3. Common internodal connections. (a) Star network; (b) basic ARPANET topology (each node is an IMP or TIP).

The customers in the ARPANET are either computers (called "hosts" in the network terminology) or terminals. Host computers access the network through an Interface Message Processor (IMP) which is a general-purpose minicomputer. Terminals access the network through either a Terminal Interface Processor (TIP) or a local host computer. Thus, at every node there is a message-switching computer. Note that the network topology is a set of interconnecting loops. Since the transmission links are bidirectional, there is a minimum of two separate routes between any two nodes, which increases reliability. Because of the statistics of the input traffic, the average message size is small. However, no matter what the size of the offered message, the IMP breaks it into small **packets** of approximately 1000 bits maximum size. Each packet becomes a small message in itself, to which the IMP appends addressing and other information for network use (e.g., linkage information for reassembly of multipacket messages and error control bits). The maximum throughput of an IMP, where throughput is defined as the sum of the message bits entering and leaving the IMP each second, is approximately 700 kilobits per second.

As a packet travels through the network, each node stores it until it receives from the next node an acknowledgment that the packet has been received correctly. The route a packet follows is not determined in advance. At each node the IMP will select a minimum delay route to the destination. This requires that each IMP have a knowledge of the queue lengths at every other node. The information is stored in routing tables, which are dynamically updated approximately twice a second. Route selection is thus by table lookup. Since routing is adaptive, packets in a multipacket message may travel by different paths and arrive at the destination out of sequence, which necessitates linkage information in the packet header.

As a result of this innovative use of computers, very impressive operational results have been achieved. Trans-network delays have been of the order of tens of milliseconds (depending on message lengths), well under the desired 0.5 sec. Reliability has been brought to a high point, and the cost per megabit transmitted is lower than any other method except mailing a computer tape.

The ARPANET has incorporated a satellite link between the network in the continental United

States and Hawaii. Because of the much longer propagation delay, standard packet-operating procedures have had to be modified, using essentially the same fundamental techniques that have proved their efficiency for satellite links.

Today the most expensive part of a communication network is the cost of the lines. With the cost of minicomputers being rapidly reduced, very substantial savings can be effected by using the computers to concentrate the traffic at a point close to the originating terminals. Advantage is taken of the statistics of the incoming traffic to increase the utilization of the long-haul lines by buffering and sophisticated operating procedures. Concentrator requirements, which once posed a rather large design effort, now are fulfilled by off-the-shelf minicomputers.

The final major application of computers in communications is in the circuit switch, the classical telephone central office. Fig. 4 is a functional block diagram of a typical circuit switch. The switching matrix is the element that actually makes the circuit

connection between subscribers. It is a highly specialized device, either electromechanical or (today) solid state. The computer task is to control the action of the matrix. Status of the lines and trunks (such as line busy, idle), changes in state (requests for service, etc.), and control signals (dial signals, multi-frequency tone signals) are monitored by line-scanning equipment at a rate sufficiently high to insure that every line is scanned sufficiently often so that no information is lost. This information is then transferred to the computer, which assembles it, interprets it, decides what must be done, and then causes the control elements of the matrix to execute the required action.

A typical sequence of actions for the computer upon detecting a request for service ("off hook") would be (1) to look in memory for an idle path to a dial tone generator and order the matrix to connect the path, (2) upon receiving dial pulses to assemble them, (3) to decide whether the call is local or not, etc. A moment's consideration of the complexities involved in making even a crosstown call will

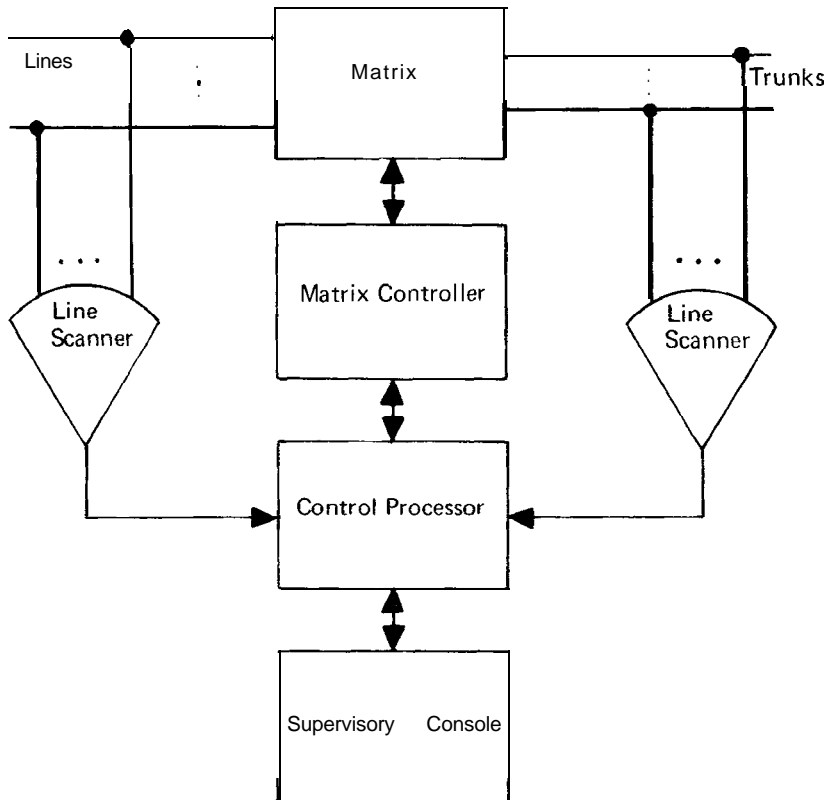


Fig. 4. Block diagram of a stored program circuit switch.

COMPATIBILITY

indicate why such a very large number of sequences of actions must be performed. However, because each action is the result of a fixed logical sequence of decisions, the digital computer is ideally suited to the job of controlling the switching center.

The computational load imposed upon the computer is twofold. First there are a large number of highly repetitive, fast response computations associated with status monitoring; e.g., line scanning. The logical level of these tasks is low, but the repetition rate is high and to a large extent independent of the number of calls in the switch. Secondly, the other tasks require a much higher level of logical decision making: routing, addressing, path hunting, etc. These tasks are highly call-dependent. In many successful systems the same computer handles both loads. However, even though each individual status monitoring action requires negligible computation, the high repetition rate and large numbers (up to 10,000 lines to be scanned in 100 ms in a large machine) of interrupts imposes a severe overhead load on the computer. For this reason the modern trend is toward the use of multiprocessors, with one processor performing the repetitive fast response chores and the second the higher-level decisions and overall control. Both processors share a common memory. It is likely that in future designs the multiprocessor configuration will be pushed further and the switch configuration will come to reassemble to some extent the current electromechanical switches, with minicomputers replacing the various markers, senders, and translators.

The stored program has been a mixed blessing for circuit switching. All the hoped for advantages of flexibility, new features, better control, etc., have been realized. However, software has been a serious problem. Because of the need for high running efficiency, all programming for circuit switching is done in assembly language or even machine language. No higher-level language has been successfully used in circuit switching. But because of the efficiencies achieved, cross-office connect times of 100 to 200 ms have been achieved today, and connect times of 10 to 50 ms are not out of line for the near future. However, these numbers should be regarded with some caution, since they are highly dependent upon the traffic loading of the switch.

REFERENCES

1967. Feller, W., *An Introduction to Probability Theory and Its Applications*, vol. 1, 2d ed. New York: Wiley.

1967. Hillier, F. S., and G. J. Lieberman. *Introduction to Operations Research*. San Francisco: Holden Day.

H. A. HELM

COMPATIBILITY

For articles on related subjects see **EMULATION**; **MACHINE INSTRUCTION SET**; **MEMORY**; Auxiliary; **OBJECT PROGRAM**; **SIMULATION**; Principles; **SOFTWARE**; and **SOURCE PROGRAM**.

Compatibility is a term applied to both hardware and software systems to describe the ease with which a computer program running on one machine may be made to run on another machine. Hardware compatibility is achieved through similarity of instruction sets (or the ability to simulate similarity of instruction sets), whereas software compatibility deals with the use of a language that can be translated into the (perhaps very different) instruction sets of several machines. Compatibility in both directions between two machines is rarely of interest, however. Usually, interest is centered on the transfer of programs and systems from a particular computer to another particular computer (and only in that one direction). Two computers are said to be *hardware compatible* if the object code from one machine can be loaded and executed on the other to produce exactly the same results. One way of achieving this is by using identical instruction sets in both machines. Another way is the use of special instructions in the second machine so that it recognizes instructions which it does not have (or which it does have but will not produce the same result when executed) in the code being transferred. These instructions are then translated into instructions on the second machine, which produce results identical to those of the first machine.

Two computers are said to be *software compatible* with respect to a particular language if a source program from one machine in that language will compile and execute to produce acceptably similar results on the other.

But since one-way acceptable compatibility is of more interest than exact intercompatibility, the terms "upward" and "downward" compatibility have come into common use. *Upward compatibility* refers to the amount of similarity of a newer, bigger,

or better (hence, upward) computer compared to that of a smaller one, but this applies only to the transfer of programs *from* the older one *to* the newer one. Similarly, “downward” compatibility refers to the transfer from the newer one to the older one.

Upward compatibility refers not only to computers with respect to the programs that run on them, but also to the data that they accept and operate on. For example, a computer software system is said to be upward compatible if identical data will produce identical results on a more recent (hence, upward) version as on an older version, even though the newer version may also accept additional forms of data. The term “identical” in this context is somewhat utopian because it almost never is realized in practice.

Manufacturers have historically extolled upward compatibility as an improvement of their small machines extended to their own larger machines, while minimizing any compatibility (especially upward) between their machines and those of their competition. However, they have been quick to point out the upward compatibility of their equipment as compared with that of the competition. In fact, computing equipment and compilers of particular manufacturers have been deliberately designed so that programs running on competitive equipment can be easily converted to run on their systems. Conversely, equipment and systems have also been designed to maximize the difficulty of converting programs so that they cannot be run on competing equipment or systems. The result has been that true compatibility is almost never achieved between equipment from different manufacturers.

Hardware component compatibility is another area where competitive practices have been counterproductive. Since many peripheral devices are hooked to the computer by a relatively small number of cables (usually with a plug, in fact), so-called plug-to-plug compatible peripherals have been developed by some competitive firms. Their practice is to purchase, say, a tape drive, find out how it works (spending only a fraction of the original development costs), and build one that works exactly the same (and even has identical plugs on the ends of the cables) as the original, but which can be profitably marketed at a much lower price than the original. Thus, potential customers exist wherever the original equipment was installed.

Manufacturers have developed several defenses against these practices. Probably the most compelling deterrent is refusal to provide a maintenance contract on a system in which parts have been supplied by a competitor. The implication of this

policy is, of course, that maintenance will be done by the **hOUr** (rather than on a flat-fee basis, as is usual with most contracts) and that sufficient service time will be spent to off**se**t most of the user’s savings earned by installing **co**mpetitive equipment. This substitution practice has been declared illegal in several cases, and has been rendered less effective recently by the mass conversion (where price **advan**-tages show) to plug-to-plug compatible peripherals by the federal government (by far the largest owner and lessor of equipment in the computer world). It has also been countered by the original designers, who have designed peripheral equipment wherein the most expensive part (called the “controller”) is integrated (wired directly) into the central processor, leaving only the relatively less expensive mechanical part of the device to be a target for substitution by plug-to-plug compatible replacement.

Although integrated equipment serves to deter replication of parts by competitors, this approach has run into legal complication. In a recent landmark suit, IBM was held to be indulging in monopolistic practices by implementing this integration, despite the fact that the plaintiff in the case, Telex, was found guilty of stealing trade secrets when it built a plug-to-plug device (with controller) that made interchange possible. The verdict against IBM was later overturned on appeal, and was then settled by the litigants on terms favorable to IBM before reaching the U.S. Supreme Court. This result is expected to have far-reaching effects throughout the computing industry.

C. L. **MEEK**

COMPILE AND RUN TIME

For articles on related subjects see **LAN**-**GUAGE PROCESSORS**; **OBJECT PROGRAM**; and **PROCEDURE-ORIENTED LANGUAGES**, **PROGRAMMING IN**.

The complete process of running a program that has been written in a higher-level language such as **Fortran** or **Cobol** is accomplished in two steps:

1. Translation of the source program as written by the programmer into a machine executable form (a process commonly referred to as “compilation”).
2. Execution of the generated form; i.e., the *running* of the compiled or *object* program.

COMPILER

To distinguish between certain actions that may occur during one or another of these phases, the period of compilation is known as the "compile time" and the succeeding period as the "run time." In the usual compile and execute system, these two phases are distinct and may be temporally separated. In fact, the running of a program may be accomplished many times without the need for the re-compilation of the program, provided the compiled code is saved on tape, disk, or (occasionally) punched cards. In an interpretive system, however, the two phases are intertwined, since execution of each piece of source program follows immediately after its "compilation".

Typically, errors in a program are related to compile time or run time. Where the error is an error of language (i.e., incorrect syntax such as a missing parenthesis), then the system is capable of recognizing this at compilation time; on the other hand, errors in logic or arithmetic (i.e., semantic errors) are normally discovered (if at all) at run time. Some sophisticated language processor systems allow the programmer to use certain facilities called *compile-time* and *run-time* facilities. As an example of the latter, some systems allow the programmer to specify the format of his input data and output results at run time rather than in the source program.

J. A. N. LEE

COMPILER. See **COMPILER, INCREMENTAL; COMPILER, SYNTAX-DIRECTED; LANGUAGE PROCESSORS;** and **LOAD-AND-GO COMPILER.**

COMPILER, INCREMENTAL

For articles on related subjects see **COMPILER, SYNTAX-DIRECTED;** and **TIME SHARING.**

The advent of conversational time-sharing systems, in which the problem-solving process invokes a dialog between the user at a terminal and a remote computer, has led to the development of various compiling techniques that can be of particular benefit to the time-sharing user. One of these is *incremental* compiling in which the compiler generates code for a statement, or group of statements, which

is independent of the code generated for other statements.

Provided the language statements entered by the user are ordered in a standard fashion, the compilation process is closely related to that used in batch-processing operations. On the other hand, if the user is permitted to present the statements in any order., such as specifying array dimensions following the usage of an array element in a statement, then more sophisticated techniques of compilation are required. In any case, the advantage of incremental compiling to the user is that he may compile and test parts of his program as he "composes" it at the terminal rather than being required to postpone the debugging process until the entire program has been written.

J. A. N. LEE

COMPILER, SYNTAX-DIRECTED

For articles on related subjects see **CUM-PILER, INCREMENTAL; GRAMMAR, GENERATIVE;** and **LANGUAGE PROCESSORS.**

A syntax-directed compiler (sometimes called a "syntax-oriented" compiler) is a general-purpose compiler that will service a family of languages by providing the syntactic rules for language analysis in the form of data, typically in tabular form, rather than building the specific parsing algorithm for a particular language into the compiler. In this manner, a single processor can be used for the compilation of many differing languages, provided only that the syntactic rules of each language can be expressed in the required format of the data. Table 1 is an example of a part of a simplified syntax table that might be used by a syntax-directed compiler. To see how such a table is used, consider the entries 2.0-2.3. If the compiler is searching for the construct **TERM**, line 2.0 says that if it finds a factor (**FT**) without a following (successor) construct, this is acceptable (**OK**) but, if not, there is the alternate 2.1, which states that a factor followed by a multiply operator (2.2) followed by another factor (2.3) is **OK**, but that any other alternate fails. A syntax table, therefore, acts like a series of small programs, Note in particular that the syntax can be changed merely by changing entries in the table. For example, to allow a term to have the form $FT \div FT$ where \div indicates division by integers, the "fail" in line 2.3

Table 1. Part of a syntax table

Language Construct	Index	Name	Successor	Alternate
Arithmetic	1.0	TE	1.1	Fail
Expression (AE)	1.1	AO	1.2	Fail
	1.2	TE	OK	Fail
Term (TE)	2.0	FT	OK	2.1
	2.1	FT	2.2	Fail
	2.2	MU	2.3	Fail
	2.3	FT	OK	Fail
Factor (FT)	3.0	PR	OK	3.1
	3.1	PR	3.2	Fail
	3.2	EO	3.3	Fail
	3.3	PR	UK	Fail
Primary (PR)	4.0	c o	OK	Fail
	4.1	UA	OK	Fail
Exponentiation operator (EU)	5.0	**	OK	Fail
Add operator (AO)	6.0	+	OK	6.1
	6.1		OK	Fail
Multiply operator (MO)	7.0	*	OK	7.1
	7.1	/	OK	Fail

could be replaced by “2.4” and lines “2.4–2.6” would contain the entries for **FT**, \div and **FT**.

Syntax-directed compilers also form the basis for a compiler-compiler, which is a specialized processor that generates compilers (McKeeman, 1970).

REFERENCES

1966. Ingerman, P. Z. *A Syntax Oriented Translator*. New York: Academic Press.
1970. McKeeman, W. M. *A Compiler Generator*. Englewood Cliffs, N.J.: Prentice-Hall.

J. A. N. LEE AND A. RALSTON

COMPLEMENT

For articles on related subjects see **ARITHMETIC**, **COMPUTER**; and **NUMBERS AND NUMBER SYSTEMS**.

In ordinary arithmetic we represent negative numbers by a minus sign followed by the absolute value (i.e., magnitude) of the number (e.g., -6.42). In computers we can represent negative numbers this way also, and sometimes this is actually done, but more often a “complement” representation is used.

Even when the sign-magnitude representation is used, the hardware of the computer normally will include a **complementer** to assist in carrying out the various arithmetic operations.

To motivate the need for complements or complementers, consider the addition of two numbers expressed in sign-magnitude form. Before the operation can be carried out, the signs of the numbers must be compared. If they are the same, the two numbers can be added; if they are different, the smaller in magnitude must be subtracted from the larger and the correct sign appended to the result. As we will see, the use of complements avoids much of this complication.

Definitions. There are two kinds of complements, *radix* complements and *diminished radix* complements, where “radix” refers to the base of the number system being used. Let x be a positive number in the decimal system. Then the diminished 10’s complement of x , which we denote by \bar{x} and which is generally called the “9’s complement,” is formed by subtracting every digit of x from 9. Thus, if $x = 426.3091$, $\bar{x} = 573.6908$. The 10’s complement \tilde{x} is defined as the result of adding 1 in the least significant place of \bar{x} or, equivalently, as the result of subtracting x from 10^n , where n is such that the 1 in 10^n is one place to the left of the most significant digit of x . Using the above example, $\tilde{x} = 573.6909 = 1000.0000 - 426.3091$. Both the quantities \bar{x} and \tilde{x} are thus representations of the quantity $-x$.

COMPLEMENT

The other complements of practical importance are those in the radix 2, or binary, system. If x is now a positive binary number, then its “1’s complement” \bar{x} is formed by changing all 0’s in x to 1’s and 1’s to zeros (i. e., subtracting all bits of x from 1) and the “2’s complement” \tilde{x} is formed by adding 1 in the least significant place of \bar{x} or, equivalently, subtracting x from 2” with n chosen as above. Thus, if $x = 10.1101$, then $\bar{x} = 01.0010$ and $\tilde{x} = 01.0011 = 100.0000 - 10.1101$.

Properties of Complements. The useful properties of complements in computers are best illustrated using the binary system. For illustrative purposes consider a computer where the numbers on which arithmetic operations are to be performed each have eight bits, the first of which denotes the sign (0 for plus, 1 for minus) and the other seven bits are, for convenience, assumed to represent an integer. If the sign is negative, let us assume the integer is in the 2’s complement form. Then, to add two such numbers, we need only treat them as eight-bit positive integers (i.e., treat the sign as another bit of the number), add them, and discard any carry to the left of the eighth position (see Fig. 1). Thus, we are able to ignore both the sign and relative magnitudes of the two numbers. With negative numbers in the 1’s complement form, there is the slight additional complication that carries to the left of the eighth position must be added into the first (i.e., least significant) position (see Fig. 2).

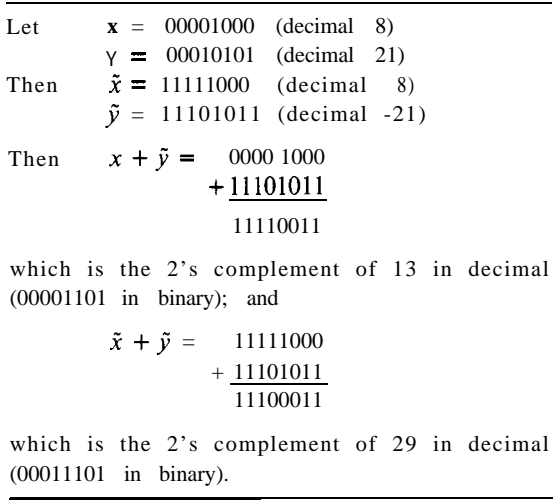


Fig. 1. Addition of numbers using 2’s complements.

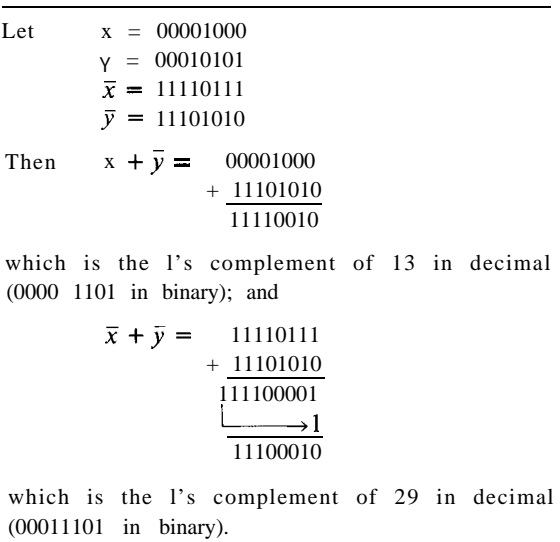


Fig. 2. Additions of numbers using 1’s complements.

Both results given above are rather easily proved by writing complemented numbers as 2^n minus the corresponding positive number (minus 1 for 1’s complements). One interesting property of the 1’s complement form is the existence of two zeros, one with a positive sign and one with a negative sign. This follows because the 1’s complement of 00000000 is 11111111. With 2’s complements, however, there is only one zero, since the 2’s complement of 0000 0000 is 10000 0000, which has nine bits. In 2’s complement representation, 1111 1111 is the complement of 00000001. The existence of two zeros can be used with advantage in some contexts, but requires a somewhat more difficult test to determine if a number is zero than would otherwise be the case. Since 1’s complements are generated merely by changing 0’s to 1’s, and vice versa, it is very easy to build a circuit to generate the 1’s complement of a

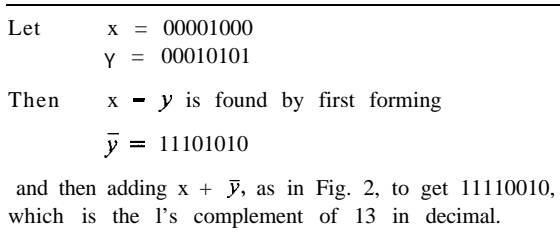


Fig. 3. Subtraction using 1’s complements

number. It is somewhat more difficult, but not very hard, to build a circuit to generate 2's complements. Therefore, it is easy to perform subtraction by first complementing the minuend and then adding (see Fig. 3). This means it is not necessary to have a hardware subtracter if there is a hardware adder and a complemeter. It is for this reason that computers rarely have subtracters.

For performing multiplication and division, there are no direct advantages to the complement form and some disadvantages. However, the adjustments to algorithms for multiplying or dividing two positive numbers to allow them to handle operands in complement form are not major. Alternatively, negative operands in multiplication or division can first be complemented and then the appropriate sign can be appended at the end.

Most modern computers store negative numbers in either 1's or 2's complement form. Which of the two forms to choose depends upon some rather subtle and by no means conclusive considerations concerning the details of computer circuitry. Even in those rather rare cases when negative numbers are stored in sign-magnitude form, it is usual to have a complemeter in the arithmetic unit for use in performing arithmetic operations involving negative numbers or subtraction.

A. RALSTON

COMPLEXITY. See COMPUTATIONAL COMPLEXITY.

COMPUTABILITY

For articles on related subjects see **ALGORITHMS, THEORY OF**; and **DECIDABILITY**.
For article on related term see **ALGORITHM**.

"Computability" is a property of **functions**. A function f with domain D and range R is a definite correspondence by which there is associated with each element x of the domain D (referred to as the "argument") a single element $f(x)$ of the range R (called the "value"). The function f is said to be computable if there exists an algorithm that, for any given x in D , provides the value of $f(x)$.

For example, consider the function g , whose domain D is the set of all pairs of positive integers and whose range R is the set of positive integers, and which is defined as

$g(a,b)$ = the greatest common divisor of a and b .

This function is computable by the well-known Euclidean algorithm (**Knuth**, 1968).

The above definition is lacking rigor for the following reasons:

1. It is not explained in what form the argument x in D is "given." In particular, this part of the definition makes sense only if elements of D are in some way finitely describable. Thus, the notion of computability, as described above, makes no sense for a function f whose domain is the set of real numbers.
2. The notion of an algorithm is not precise.
3. It is not explained in what sense the algorithm provides us with the value of $f(x)$.

How the notion of computability of a function can be made mathematically rigorous is explained in the article, **ALGORITHMS, THEORY OF**.

REFERENCE

1968. Knuth, D., "The Art of Computer Programming," in Vol. 1, *Fundamental Algorithms*. Reading, Mass. : Addison-Wesley.

G.T. HERMAN

COMPUTATIONAL COMPLEXITY

For articles on related subjects see **ALGORITHMS, THEORY OF**; and **TURING MACHINE**.

For articles on related terms see **FAST FOURIER TRANSFORM**; **FORMAL LANGUAGES**; and **MATHEMATICAL PROGRAMMING**.

The complexity of computations must have been an issue since the time counting came to play a role in human culture. Current widespread use of modern digital computers has substantially increased the importance of questions about the difficulty of computational problems.

COMPUTATIONAL COMPLEXITY

Obvious and familiar algorithms are continually being replaced by more efficient ones as programmers and mathematicians gain more sophistication in computing methods. The aim of the study of computational complexity is to develop techniques for discovering better algorithms and to explain why some computational tasks are difficult, no matter what algorithm is applied to them.

The Computational Complexity of Multiplying Integers. Computing the product of two integers is an example of an apparently simple problem whose computational complexity turns out to hold some surprises.

One natural way to measure the difficulty or computational complexity of multiplication is to count the number of basic operations on digits—such as reading or writing a digit, adding two digits, or multiplying two digits—required to multiply two integers. The usual grade-school method for computing the product of any two n digit integers requires each digit of the multiplier to be multiplied by each digit of the multiplicand, so the number of basic digital operations required is proportional to n^2 .

In contrast, the sum of two n digit numbers, again using the familiar grade-school algorithm for addition, can be calculated with a number of digital operations proportional to n . This latter algorithm is surely close to optimal, since proportional n operations are necessary just to read the numbers to be added.

Experience with these simple arithmetic algorithms suggests that multiplying is harder than adding; adding is even used as a subprocess in grade-school multiplication. While this operation is well suited for calculation with pencil and paper, and may also have a certain nostalgic appeal, the grade-school multiplication algorithm is far from the most efficient algorithm now known. Over the past decade a succession of faster methods have been discovered, culminating in a method that requires only proportional $n \cdot \log n \cdot \log \log n$ digital operations. This fast method is based on an algorithm known as the “Fast Fourier Transform,” which itself performs in $n \cdot \log n$ operations—a computation which, like multiplication, at first sight seems to require n^2 operations.

There is no record of fast multiplication methods having been discovered earlier than 1962. Considering how many of history’s greatest mathematicians were also prodigious calculators, it is remarkable that fast multiplication should be such a recent discovery. (One explanation might be that the bookkeeping details involved in some of the highly

optimized methods are too complicated for hand calculation and demand the patient reliability of computing machines.)

Whether multiplication really is harder than addition is still unknown. There is no reason to suppose that the best way to multiply has already been discovered, and it may be that multiplication of n digit numbers can be performed in proportional n operations. On the other hand, if the most efficient method presently known is the best, proving that it is will be an interesting mathematical challenge. Theoretical results about computational complexity reveal the possibility that there might not even be a best way to multiply!

The Mathematical Theory of Computational Complexity. The theory of computational complexity provides a mathematical framework for studying all algorithms that accomplish any computational task. A difficulty faced by this study is that, for any computable mathematical function (of which multiplication is only one of the simplest examples), there are an infinite number of different ways to compute it. Anyone with a bit of programming experience will realize that there is endless variety possible in programs for any given task; this observation can actually be proved as one of the elementary theorems of the mathematical theory.

Mathematical theory begins with the notion of an algorithm or program that computes a function $f(x)$ on the integers: Given any input integer x , the algorithm is applied and yields as output the integer $f(x)$. Algorithms can be defined rigorously as being programs for simple models of computers such as Turing machines. The general properties of algorithms and computable functions have been studied by mathematicians for nearly 40 years, and it is generally agreed that anything which could reasonably be regarded as an algorithm can be carried out by a Turing machine.

The time required by an algorithm A depends on the input to which A is applied. The complexity of A is therefore defined to be the function $T_A(x)$ equal to the number of basic steps required by algorithm A applied to input x . Actually, the number of steps is just one measure of complexity and other measures, such as amount of storage space or number of memory accesses, may also be used.

Two axioms characterize most of the basic properties of complexity measures:

1. $T_A(x)$ is finite if and only if algorithm A applied to input x eventually halts and gives an output. (In other words, an algorithm halts if and

only if it halts after a finite number of steps.)

2. There is an algorithm which, given as inputs any integers x and y and any algorithm A , will determine whether or not $T_A(x) = y$. (Given x , y , and A , one can simulate A applied to x for exactly y steps and see whether A halts on the last step.)

These straightforward axioms are enough to imply, for example, that there are computable functions which cannot be computed rapidly by any algorithm, and that more functions can be computed if more time is allowed. They also imply a much less obvious fact, known as the "Speed-up Theorem": There is a computable function f with the property that given any algorithm A which computes f , there is another algorithm B which computes f "much faster" than A . "Much faster" is interpreted by choosing any rapidly growing computable function such as 2^x ; then, according to the speed-up theorem, there is a function f such that if A is any algorithm for f , there is always another algorithm B for f such that $2^{T_B(x)} \leq T_A(x)$ for all large integers x . Thus, algorithm B requires at most the logarithm of the time required by A .

Of course, since B is itself an algorithm for f , there must be another algorithm C for f which requires only the logarithm of the time for B , and so on. Clearly, there is no single most efficient way to compute such an f .

Also, notice that f must be hopelessly difficult to compute even though it has faster and faster programs. Each program for f must require more than 2^x , and more than 2^{2^x} , and so on, steps for all large inputs x ; otherwise, the program could only be "sped-up" by an exponential a fixed number of times before "hitting bottom," after which it could not be sped up further.

These conclusions may seem to violate intuition, but they follow inevitably for any model of computer with any reasonable notion of computational step. The speed-up theorem is proved using "diagonal" arguments similar to those used in the theory of algorithms to establish the existence of undecidable problems.

Inherently Complex Computation Problems. The first uncontrived examples of functions with computational properties similar to those predicted by the speed-up theorem were discovered in 1972. Dozens of examples have now been discovered, mainly in the areas of mechanical theorem proving and automata theory.

A mechanical theorem-proving problem of this type is to determine whether sentences written in the

standard notation of mathematical logic express true statements about addition of integers. In this notation, a statement such as "not every integer is divisible by 3" could be expressed by writing

$$\sim \forall x \exists y [y + y + y = x],$$

which literally is read "it is false that, for every integer x , there exists an integer y such that $y + y + y = x$." This statement, of course, is true. The statement "for every integer x it is false that $x + x = x$," which would be written

$$\forall x [\sim (x + x = x)],$$

is not true, since $0 + 0 = 0$.

Although not all theorems about integers can be proved automatically, the additive properties of integers are simple enough so that all sentences of this kind can be proved true or false by a mechanical procedure. One can design an algorithm which, given as input any sentence about integer addition, will correctly produce the output "True" or "False."

An efficient procedure of this kind would yield other efficient procedures for linear and integer programming problems and a variety of similar optimization problems with immediate practical applications, since these mathematical programming problems involve only additive properties of integers. Sentences about integer addition turn out to have enough expressive power that, if one hypothesized that their truth could be decided quickly, it would follow that any problem decidable within 2^n steps could also be decided quickly. Abstract complexity theory rules out the latter possibility, and it follows that any algorithm which decides the truth of sentences about integer addition must eventually, when applied to sentences of length n , use more than 2^n basic digital operations. The values of 2^n become astronomical for $n \geq 6$; any correct procedure for solving this computational problem would rapidly exhaust the resources of the known universe.

REDUCIBILITY AMONG PROBLEMS. Computational problems that appear to be complex arise in nearly every engineering and scientific discipline. A rich variety of problems in areas such as operations research, computer design, data manipulation, graph theory, and mathematical logic cannot be efficiently solved by any known procedure, but present mathematical knowledge leaves open the possibility of efficient solutions for them.

There seem to be so many kinds of computational problems that, even should many of them prove to be inherently difficult, they might be

COMPUTATIONAL COMPLEXITY

difficult for different reasons. For example, each of the following different problems can be solved by algorithms, using a number of steps that is an exponential function of the "size" of the problem, while none of them are known to be solvable by algorithms using only a polynomial number of steps:

1. Given the distances between n cities, find the shortest "traveling salesman" route which visits each city exactly once.
2. Given an arbitrary map, determine whether the countries can be colored, using only three colors so that all adjacent countries have different colors.
3. Given a circuit built of gated logic, find the smallest equivalent circuit.
4. Given a truth-functional formula, determine whether it is identically true.
5. Given a record of mutual friendships among n people, find the largest group of them such that every pair of people in the group are friends.
6. Given a set of linear arrays stored in a computer memory, find a subset of them which fills a given block of memory as fully as possible.

Despite their apparent differences, these problems are virtually the same from a computational point of view. Each one is "efficiently reducible" to any other in the sense that an efficient algorithm for any one of them could be used as a subroutine in the design of an efficient algorithm for any of the others. If one of these problems can be solved in polynomial time, then all can; and if they really are hard, they surely are all hard for the same reason.

Other groups of problems—such as matrix multiplication, finding paths in graphs, and parsing context-free languages—can also be intimately related by efficiently reducing one to another. The most efficient known parser for general context-free languages was discovered only in January 1974 by reducing this problem to the problem of computing matrix products (for which a fast algorithm has been known for a few years).

The discovery of efficient reducibilities is beginning to bring intellectual order to the diversity of computational problems by coalescing dozens of problems into a few basic ones.

CONCRETE PROBLEMS. Special computational problems like sorting, evaluating polynomials, or approximating roots of rational functions have led to the formulation of special **classes** of algorithms for which precise bounds on difficulty can be proved.

For sorting problems a natural class of algorithms involves comparisons between elements as the only basic operation. The maximum among n ele-

ments, for example, can be found with only $n - 1$ comparisons in an obvious way. Since each of the $n - 1$ elements other than the maximum must have been determined to be the smaller of two compared elements, any comparison algorithm for finding the maximum must require this many comparisons. Completely sorting n elements can be accomplished with little more than $n \cdot \log n$ comparisons, and a simple counting argument shows that this many comparisons are necessary. Finally, finding the median (middle element) seems at first to require a full sorting, but an extremely clever new algorithm does the job in only $3n$ comparisons; at least $7n/4$ are known to be necessary.

A natural complexity measure for the problem of evaluating a polynomial is the number of arithmetic operations (additions, subtractions, multiplications, and divisions) required. **Horner's rule** is a classical trick for factoring a polynomial of degree n so that only n multiplications are required to evaluate the polynomial at any given point. Better methods have been discovered which require only half this many multiplications, and the latter number can be shown by algebraic arguments to be necessary in general.

Conclusions. Proving lower bounds on complexity of computation is essential for a thorough understanding of the potential power of computing devices. When efficient algorithms for some computational problem are provably impossible, the proof can provide important clues for reformulating or specializing the problem so that efficient methods can be found. Proofs of impossibility, of course, also save time and brainpower that might otherwise be spent futilely. It is remarkable that in many cases it is possible to prove such lower bounds.

The mathematical analysis required to prove that an algorithm is optimal can also lead to the happy discovery of a still better algorithm. Some of the fast algorithms mentioned in this article, as well as several others for such problems as matrix multiplication, finding paths in graphs, testing graphs for planarity, and evaluating integer polynomials, were found after the failure of attempts to verify that known algorithms were the best possible.

Research on computational complexity has just begun, but already it has explained why some computations are hard and some are easy.

REFERENCES

- 1968, 1969, 1973. Knuth, D. *The Art of Computer Programming*, Vol. I, II, and III. Reading, Mass.: Addison-Wesley.

1973. Borodin, A. "Computational Complexity: Theory and Practice," in A. Aho (Ed.), *Currents in the Theory of Computing*. Englewood Cliffs, N.J. : Prentice-Hall, pp. 35-89.

A. R. MEYER

COMPUTATIONAL LINGUISTICS.

See PROGRAMMING LINGUISTICS.

COMPUTER, USING A

For articles on related subjects see **DEBUGGING**; **DIAGNOSTICS**; **DOCUMENTATION**; **FLOWCHART**; and **PROCEDURE-ORIENTED LANGUAGES**.

For articles on related terms see **ADMINISTRATIVE-BUSINESS APPLICATIONS**; **ALGORITHM**; **DECISION TABLES**; **INFORMATION RETRIEVAL**; and **SYSTEM CHART**.

Using a computer means getting it to do some desired piece of information processing: give an answer; store some information; show some fact, condition, or trend; control a process; produce a product. In some cases these desires are expressed to the computer in the form of a program directly by the end user, the one who wants the answer. In other situations, most of the programming has been done for him by the designer of the communication system, and he can concentrate on describing the specific data and desired output. In almost all cases, he is buffered from the computer by a "use facilitator," known variously as a programmer, systems analyst, or systems programmer, who works behind the scenes to prepare the programs and operating systems that simplify the problem of communication for the end user.

Let us consider three situations described by the phrase "using a computer," but which in each case has a quite different meaning:

A. An engineer "*uses*" a computer to solve a mathematical problem.

Meaning: The engineer writes and runs his own program, using a higher-level language, say, Fortran.

B. A court officer "*uses*" a computer to find out whether an arrested man requesting bail has a criminal record.

Meaning: He sits at a terminal attached to the computer and enters the necessary descriptive information; the desired record is displayed on the screen for him.

C. A company "*uses*" a computer to handle its accounts, financial records, billing, payroll, and inventory.

Meaning: The staff of the data processing department design and implement an integrated processing system, into which transactions are entered as they occur; it handles normal operations, makes routine decisions, and in response to inquiry makes information available to management to help them make better informed decisions.

CASE A: SINGLE PROBLEM SOLUTION; PROGRAM WRITTEN BY THE END USER. The first task of the person setting out to solve a problem is too often overlooked, or too hastily skimmed over: the necessity for acquiring a thorough understanding of the problem, including what is known, what is wanted, and what additional information may be needed.

Furthermore, since the computer must always be told what to do, the data supplied is usually not enough to describe the problem to the computer. In most cases, the computer must also be told how to solve it, by having spelled out to it the computational steps to be followed to reach the solution. Most common computer languages are therefore "procedure-oriented," i.e., designed to express this procedure to the computer.

Therefore, the next step for the problem solver must be to select a method that he feels will work in the present situation, and then to go about developing an algorithm that is an accurate and detailed description of the procedure to be followed, including rules for what to do in all possible situations that may develop.

To assist him in visualizing this procedure, he will usually draw flowcharts, ranging from the very general overall view to the explicit detailing of all steps to be followed. The flowcharts have two advantages over either a narrative description or the higher-level language program itself: Whole processes involving many steps may be summarized in a single block; and the graphic nature of the flowchart, with branches labeled according to various choices and conditions, and with arrows leading from step to step, enables the designer to see at a glance what follows what.

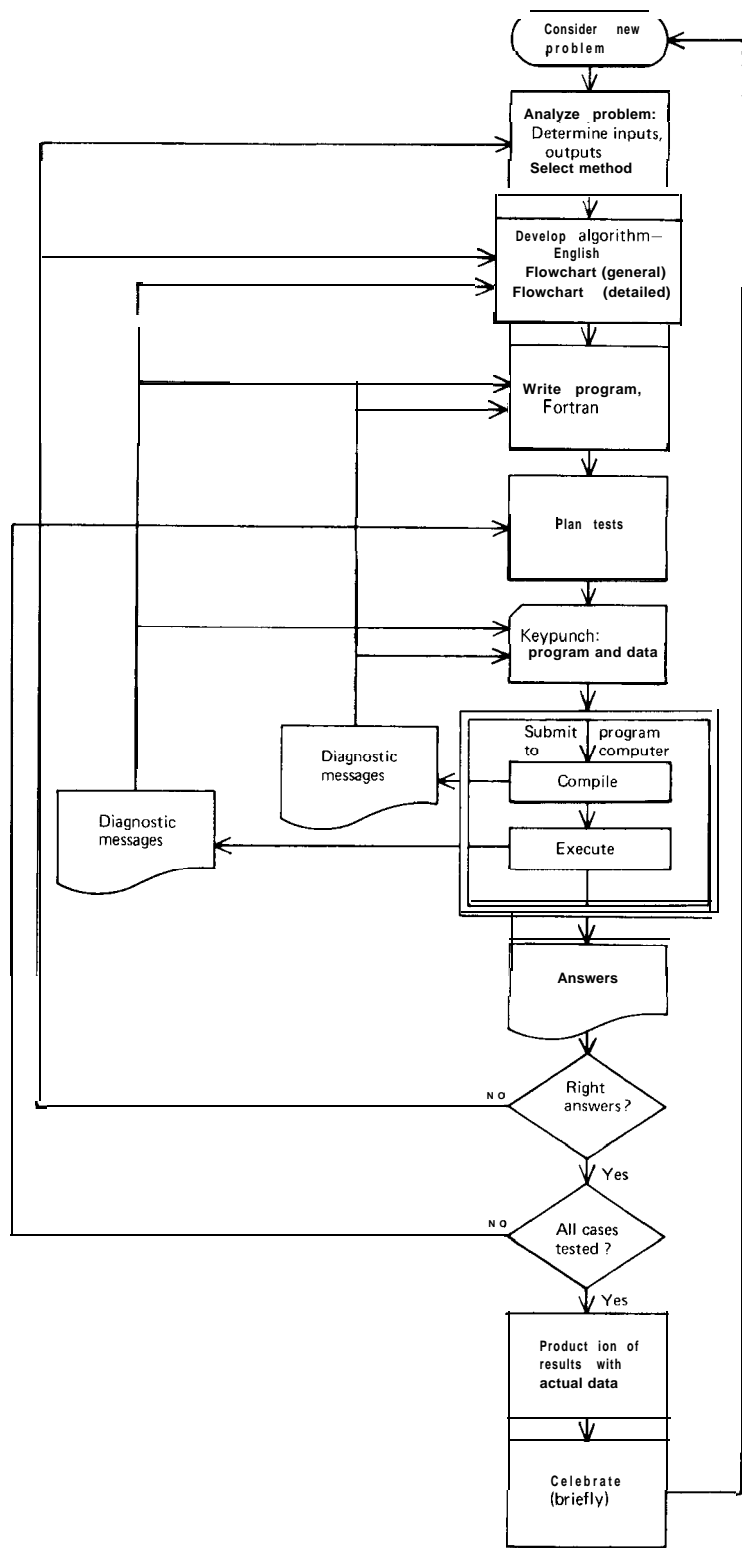


Fig. 1. Sample flowchart.

As a sample of a flowchart representation of an algorithm, Fig. 1 illustrates the procedure currently being described. Once the flowchart has been completely laid out, both in general terms and with some of the processes expanded, the programmer must then translate the algorithm from flowchart form, using arrows and general English phrases, into a language that can be accepted and understood by the computer. For scientific and engineering work, most likely this language will be **Fortran** (formula translator). Operation blocks resulting in the assignment of initial values or the calculation of new values are generally converted into series of assignment statements capable of expressing a wide variety of mathematical operations; loops and iterations are most conveniently handled by **DO** statements, which control repetition and incrementing, and branch points by **IF** statements, which test the data or various intermediate values.

As his program is being developed, a wise programmer makes plans for the inclusion of enough tests to satisfy him that the instructions are stated correctly, that the values calculated are indeed the ones he intends, and that the method works satisfactorily for the range of values with which he is concerned. These tests may involve calculating by hand the first few values or those corresponding to some special cases, examining the output values at a series of known checkpoints, or using additional mathematical or physical properties such as check sums or the constancy of certain relationships.

Furthermore, he should be realistic enough to realize that in spite of all such efforts, there will likely be logical errors and residual bugs that will never show up until certain special circumstances or numerical values occur. He can provide some further protection by building into his program some additional tests and traps, sometimes called "error exits." These may require input data to pass certain consistency and plausibility checks, may test whether certain immediate results are within a reasonable range, or may make back calculations, with the answers arrived at, to see whether they do indeed satisfy the initial equations. Failure at any of these points will then cause the program to terminate with appropriate warning messages.

In any case, these tests should be planned in advance and incorporated in the program from the very beginning. Although it is a truism that complex programs are never completely debugged, enough different cases should be chosen and sufficient variety of data selected for testing to assure some reasonable confidence in the reliability of the answers produced by the program.

Eventually, he feels that his program is complete. However, a sheet of programming statements, no matter how neatly hand-lettered or typed, is not as yet accepted as input by most computers, and the program must be converted to machine-readable form before it can be submitted to the machine. In most cases this is done by **keypunching** the program into cards, one statement to a card. Since this process, like most human operations involving routine work, almost invariably produces errors, it is advisable to verify the card deck, once punched, either by repunching on a verifier or by listing the deck to produce a printed page and comparing this with the original.

In addition to the program, the data (numerical values to be used by the program) must also be punched, as well as the control cards peculiar to the computer system being used. These control cards contain the identification of the programmer, accounting information (who is to get charged for the run), and other instructions to the operator, whether it be a human or an operating system.

Now, at last, with his deck containing control cards, program, and data, the programmer is ready to do battle with the machine. By one means or another the programmer submits his job, either by placing the cards in the card reader and pushing the button, or by giving the deck through a window to an input clerk. Depending upon the philosophy, efficiency, and busyness of the shop, and on the size of job and priority of the programmer, the turn-around time (time until he gets back a response from the computer) may vary from a few seconds to many hours.

What happens when the program is read into the computer? It must be remembered that **Fortran** (like other higher-level languages), the language in which the program is written, is not the native language of the computer. Its native language consists of a very primitive set of basic operations wired into the machine, each of which can accomplish only one small step or test one condition. Furthermore, when programming in the machine's language, the programmer must keep very careful track of exactly where every number is stored at any time. **Fortran**, on the other hand, is an intermediate or higher-level language and is much easier for people to use because they can express operations in symbols and phrases that are much closer to the way they normally describe their work. But the only way the computer can understand a **Fortran** program is first to have it translated into the corresponding series of machine-language instructions, and then execute

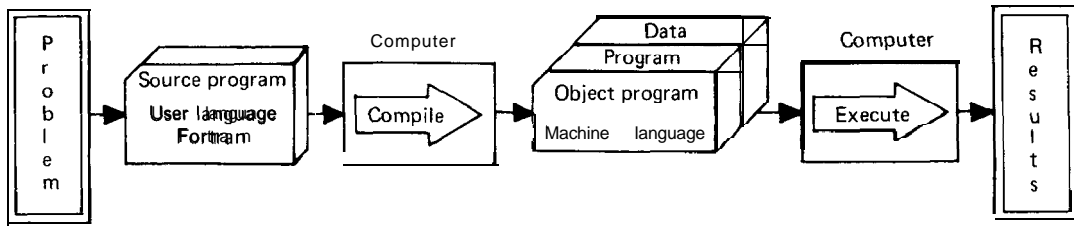


Fig. 2. Translation of Fortran to machine language. (From Davidson & Koenig, Computers, Wiley, 1967)

those. The saving feature of this process is that the translation from Fortran to machine language can be carried out by the machine itself, using an automatic translating program called a *compiler*. The process, then, undergone by the program when it is submitted to the computer can be depicted as in Fig. 2.

When the output finally is returned to the user, including the computer printout together with the original deck, the problem solver looks to see how far his program has progressed and examines the nature of the output. Human beings being what they are, it is extremely unlikely that satisfactory answers will be produced the first time, and it is usual to make from two or three to a great many iterations of this procedure, tracking down and fixing errors in the program, before the programmer is finally satisfied and the answers obtained are accepted as the ones desired.

In rare cases, our problem solver may be quite sure his program will never be needed again and may throw it away, once he has obtained his answers. More commonly, however, it will be needed for a series of runs, either by himself or by others, and it can either go directly into production or be filed in the program library. In either case, it must be properly documented while its structure and usage are still fresh.

Example (Case A): Engineer Solving a Mathematical Problem

Problem: Solution of quadratic equation

$$Ax^2 + Bx + c = 0.$$

Inputs: Several sets of three coefficients: A, B, and C.

Outputs desired: Two values of x representing the roots for each A, B, and C.

Method: Quadratic formula:

$$x = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

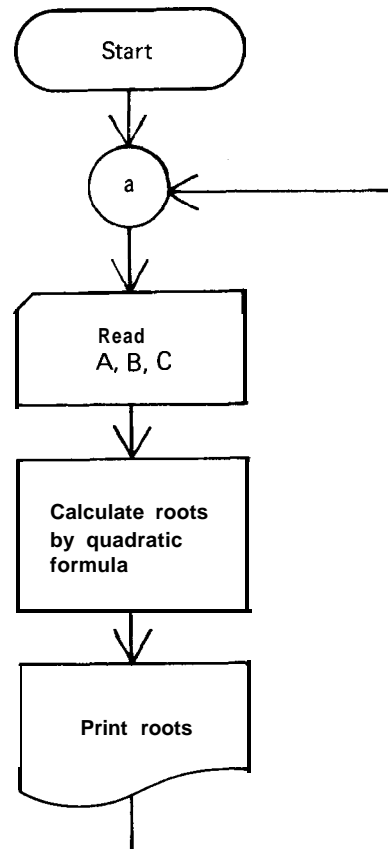


Fig. 3. Flowchart for Version 1.

This first flowchart is shown as Fig. 3. Such a flowchart can be converted into a Fortran program (Fig. 4) to carry out the various steps indicated. Test data can be easily generated by choosing special roots and working backward to determine the coefficients of the corresponding quadratic equations. For example, the roots

$$x_1 = 1 \quad x_2 = 2$$

Program

```

1 READ, A, B, C
  PRINT, A, B, C
  D = B*B - 4*A*C
  S = SQRT(D)
  X1 = (-B + S)/(2*A)
  X2 = (-B - S)/(2*A)
  PRINT, X1, X2
  GO TO 1
END

```

Data

```

1    -3    2
1     5    6
1     1    1

```

Results

```

1.0000    -3.0000    2.0000
2.0000     1.0000
1.0000     5.0000    6.0000
2.0000    -3.0000
1.0000     1.0000    1.0000

```

ERROR IN *MAIN* AT STMT. NO. 00001 + 03 LINES:
 SQRT OR DSQRT OF A NEGATIVE NUMBER
 ABORT

Fig. 4. Program for Version 1.

can easily be shown to lead to the equation

$$x^2 - 3x + 2 = 0,$$

for which the coefficients are

$$A = 1, \quad B = -3, \quad C = 2.$$

However, the unwary engineer who implements this simple flowchart, even though he checks it out with some valid test cases such as the set above, is in for some rude surprises if he tries it out on real data.

To begin with, this method finds only real roots; if for a given set of coefficients the discriminant D comes out negative, then the program will be asking for the square root of a negative quantity. In most Fortran implementations this will cause the job to be thrown off the machine, usually with a comment as to just what error occurred. (Unfortunately, some Fortran compilers may make an assumption about what you might have meant to say rather than what you did say, make an adjustment such as taking the absolute value of D , and then proceed with never a warning!) The solution is for the programmer, then, to insert a test statement before the square root step,

testing the sign of the discriminant himself so that he can specify what he wants done if it turns out to be negative. If it is, then the roots are complex, and can be expressed as

$$x_1 = \frac{-B}{2A} + i \frac{\sqrt{4AC - B^2}}{2A},$$

$$x_2 = \frac{-B}{2A} - i \frac{\sqrt{4AC - B^2}}{2A}.$$

He can then incorporate these two procedures into one flowchart, shown as Fig. 5.

This procedure will lead to a program (Fig. 6) that will handle many more cases than before, but it still will not correctly handle all cases that may be encountered. Suppose, due either to the nature of the equation or to a keypunching mistake, a set of coefficients is encountered in which $A = 0$ but both B and C are nonzero. When the step is reached in which the computer is asked to divide by zero, the program will again stop. Once again, the engineer programmer can insert another statement in his program, testing whether or not $A = 0$. If it does, then the equation reduces to a simple linear one:

$$Bx + C = 0,$$

which has the solution

$$x = -C/B,$$

provided, of course, A and B are not both zero! To be completely sure, he should also test for this, for if both A and B are zero, the equation reduces to $c = 0$, which is either a contradiction (if $C \neq 0$) or a useless truism (if also $C = 0$).

A complete algorithm for handling all these situations can be represented then by the final flowchart and its program (Figs. 7 and 8, respectively).

CASE B: INFORMATION RETRIEVAL VIA A TERMINAL. If an information storage and retrieval system such as a CORI (criminal offender record information) file is well designed, one need know nothing about computers or programming languages to use it. One needs only be taught a few simple steps such as the following:

1. Sign on: activate the terminal, establish communication with the computer.
2. Identify: give user name, number, password, or other security information.
3. Select: identify the file from which information is desired.

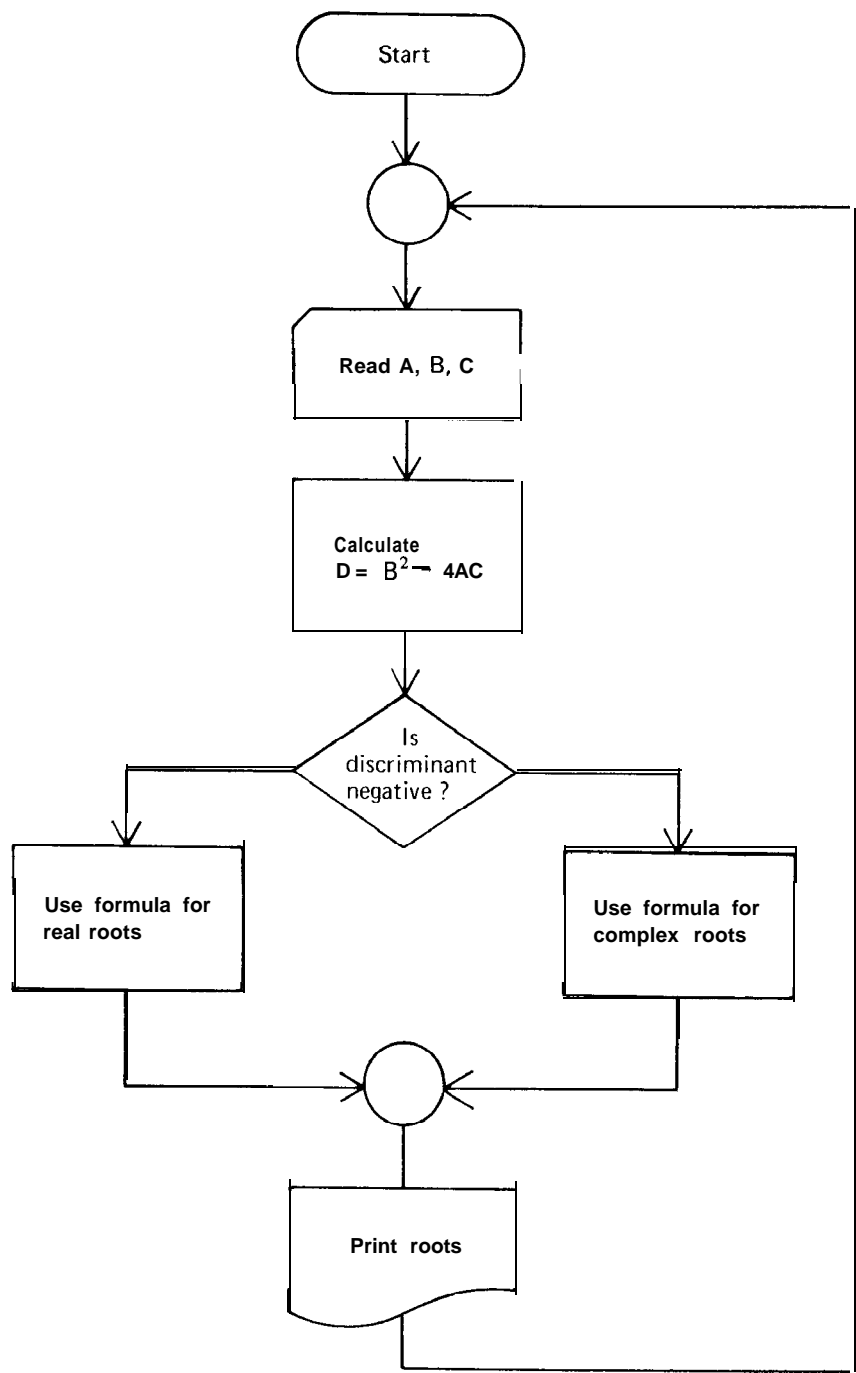


Fig. 5. Flowchart for Version 2.

```

Program
1 READ, A, B, C
  PRINT, A, B, C
  D = B*B - 4*A*C
  IF(D .LT. 0) GO TO 2
  S = SQRT(D)
  X1 = (-B + S)/(2*A)
  X2 = (-B - S)/(2*A)
  PRINT, X1, X2
  GO TO 1
2 S = SQRT(-D)
  RR = -B(2*A)
  RI1 = S/(2*A)
  RI2 = -S/(2*A)
  PRINT, RR, RI1
  PRINT, RR, RI2
  GO TO 1
END

Data
1      -3      2
1      5      6
1      1
0      3      12

Results
1 0 0000 - 3.0000 2.0000
2.0000 1.0000
1.0000 5.0000 6.0000
-2.0000 -3.0000
1.0000 1.0000 1.0000
-5.0000E-01 8.6603E-01
-5.0000E-01 -8.6603E-01
0.0000 3.0000 12.0000

ERROR IN *MAIN* AT STMT. NO. 00001 + 05 LINES:
DIVISION BY ZERO
ABORT

```

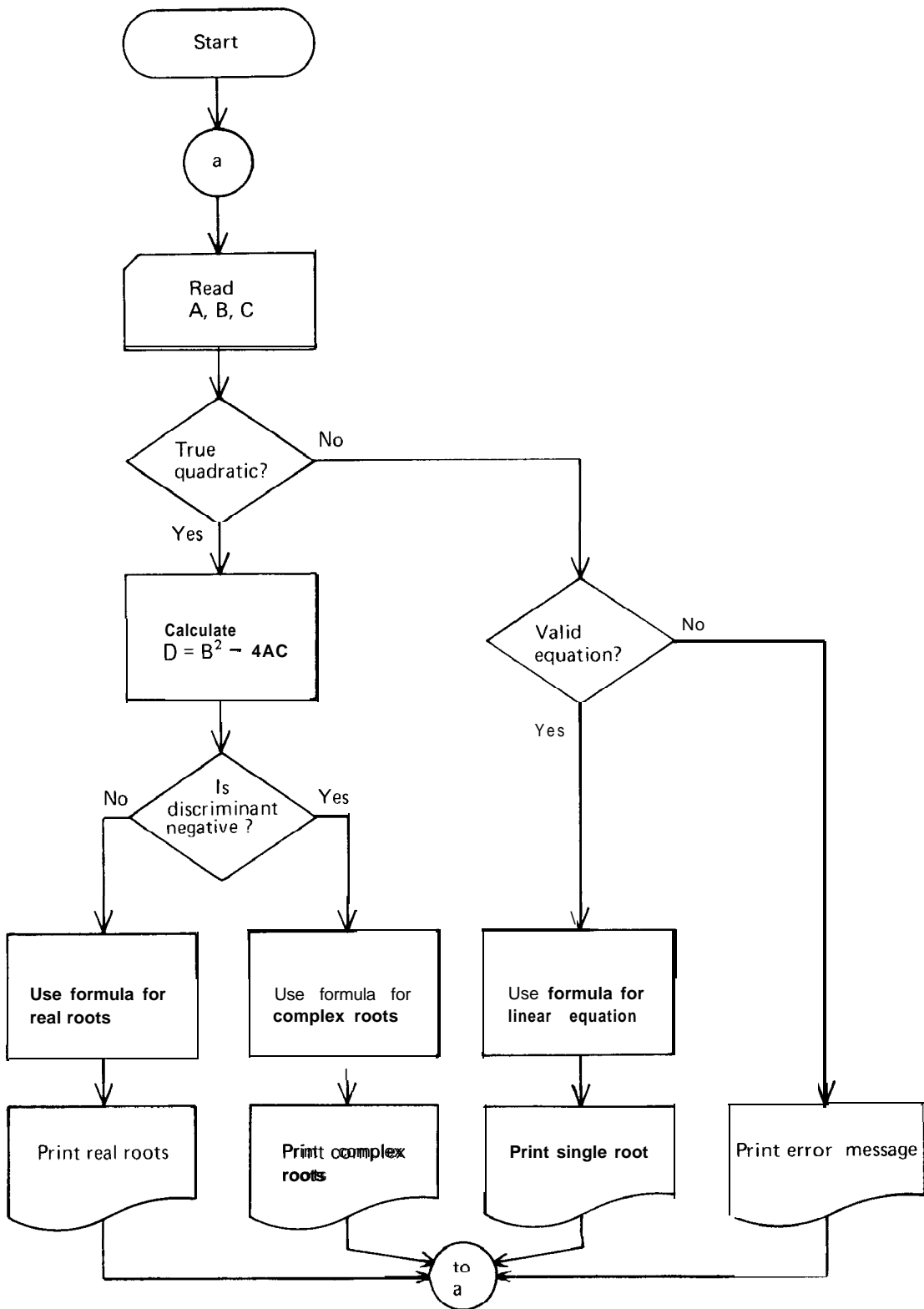


Fig. 7. Flowchart for Version 3.

```

Program
    WRITE(6,10)
    1 READ, A, B, C
      IF(A .NE. 0) GO TO 2
      IF(B .EQ. 0) GO TO 7
    C
    C EQUATION IS LINEAR
    C
      ROOT = -C/B
      WRITE(6,11) A, B, C, ROOT
      GO TO 1
    C
    C ERROR
    C
      7 WRITE(6,16) A, B, C
      GO TO 1
    C
    C TEST DISCRIMINANT
    C
      2 D = B*B - 4*A*C
      IF(D .LT. 0) GO TO 4
    C
    C EQUATION HAS TWO REAL ROOTS
    C
      S = SQRT(D)
      ROOT1 = (-B + S)/(2*A)
      ROOT2 = (-B - S)/(2*A)
      WRITE(6,12) A, B, C
      WRITE(6,13) ROOT1, ROOT2
      GO TO 1
    C
    C EQUATION HAS TWO COMPLEX ROOTS
    C
      4 S = SQRT(-D)
      RR = -B/(2*A)
      RI = S(2*A)

```

```

WRITE(6,14) A, B, C
WRITE(6,15) RR, RI
GO TO 1
10  FORMAT(6X,'A','6X','B','8X','C'/'1X,3('-----'))
11  FORMAT(3F9.2,' IS A LINEAR EQUATION'/
*      36X,'ROOT = ',1PE11.4/)
12  FORMAT(3F9.2,' HAS TWO REAL ROOTS')
13 3  FORMAT(38X,'R1 = ',1PE11.4/38X,'R2 = ',1PE11.4/)
14 4  FORMAT(3F9.2,' HAS TWO COMPLEX ROOTS')
15 5  FORMAT(36X,'REAL = ',1PE11.4/)
*      31 X,'IMAGINARY = ',1PE11.4/)
16 6  FORMAT(3F9.2,' NO EQUATION')
END

```

Data	1	- 3	2
1	5	6	
1	1	1	
0	3	12	
0	0	12	

Resu lts			
A	B	C	
1.00	-3.00	2.00	HAS TWO REAL ROOTS R1 = 2.0000E + 00 R2 = 1 .0000E + 00
1.00	5.00	6.00	HAS TWO REAL ROOTS R1 = - 2.0000E + 00 R2 = - 3.0000E + 00
1.00	1.00	1.00	HAS TWO COMPLEX ROOTS REAL = - 5.0000E 01 IMAGINARY = - 5.0000E 01
0.00	3.00	12.00	IS A LINEAR EQUATION ROOT = - 4.0000E + 00
0.00	0.00	12.00	NO EQUATION

Fig. 8. Program for Version 3.

frequently involving independent testing of each process at a time, followed by simultaneous testing of various sections working together, and then of the complete system, making sure to test it under expected full-load conditions. Even when it appears to be ready, it has been found highly advisable to plan a concurrent operation of the old system and the new for a transition period. Essential to the system at any stage are backup files and restart procedures, so that when (not if) something goes wrong, only a small amount of the most recent processing is lost and, with a minimum of redoing, all records can be completely restored.

In the data processing system case it is even

more essential than in the earlier case that complete documentation, explanation, and instructions for operation be prepared. Not only is it necessary for employees unfamiliar with either the program or the computer to use the system and to supply the daily inputs it must have, but managers, accountants, and tax investigators must all be able to gain a clear picture of how it operates, what it does, and what the meanings and validity of the results are.

Properly designed, built, tested, explained, and trained for, such a computer information processing system can be "used" by a great many people for a long time to carry out the operations of the company faster, more accurately, and more efficiently, and

COMPUTER ACCOUNTING AND RESOURCE CONTROL

above all to provide management with more meaningful information about how to run the company.

C. H. DAVIDSON

COMPUTER ACCOUNTING AND RESOURCE CONTROL

For articles on related terms **see** ADMINISTRATIVE-BUSINESS APPLICATIONS; COMPUTING CENTER; COMPUTING SYSTEMS; COMPUTING UTILITY; DATA SECURITY; FILES; and OPERATING SYSTEMS: Contemporary Features of.

For articles on related terms **see** COMMAND AND JOB CONTROL LANGUAGES; and INTERVAL TIMER.

As computer/software systems have developed, there has been a corresponding need to develop an accounting system for the resources of the system. As with any accounting system, the goal of this capability must be to charge the user for the cost of services rendered to him in such a fashion that he is motivated to evaluate the benefits of those services. The implications of this statement are that not only must there be charges rendered for these services but that also there must be means of preventing unauthorized use of the system. Furthermore, the resources used by any one user must be limited in order to prevent him from degrading the total effective services offered.

Accountability is important both to the computing center staff and to its users. In order to perform their duties as financial planners for the center, the administrators require some form of an accounting system. Such a system may be expected to yield statistics on hardware utilization and individual spending. These statistics can then be used to form the monthly and yearly reports submitted by the administrator(s) to whom the staff reports.

The completely automated accounting and resource control system—

Provides minute details concerning both hardware and software utilization along with job statistics on account spending.

Prevents unauthorized users from utilizing hardware and software facilities for which they have not received permission.

Prevents users from exceeding their allocated funds or other account limits.

Enforces job limitations on such things as page or line limits, computer time, core size, and permanent files.

Assists the operating system in providing more effective control of the resources (main memory, auxiliary memory, peripheral devices, etc.) of the system.

Development of Accounting and Resource Control Systems

EARLY COMPUTER SYSTEMS. Since early computer systems consisted of hardware with little or no software support, automated accounting was almost nonexistent. The accounting that did exist was done by user sign-up sheets, time clocks, or a flat-rate charge per computer run.

The idea of basing charges on the value of resources used was not very important either. Because the entire computer was dedicated to the current user, there was little reason to charge less if the program used only half of memory or no tapes, etc. Besides, the hardware could not usually support an accounting system because there was no hardware-readable clock and it was often not practical for the machine operator to type in the date and time for each job logged on the machine.

EARLY AUTOMATED ACCOUNTING SYSTEMS. As Rosin (1969) points out, the first accounting systems were often nothing more than system logs produced by the "on-line" printing facility. Since the purpose of such a log was to record the use of major system components, the log was more useful for measuring system behavior than for actual user accounting.

Some systems were enhanced by a hardware-readable clock, which made it possible to log the time along with the system component in use. However, the content of each entry in the log was a function of the sophistication of the resident monitor and often provided only such information as log-on and log-off time. Thus, user accounting was still based on total machine time used, with the hardware clock now providing a more accurate method for recording that time.

EXECUTIVE SYSTEMS AND AUTOMATED ACCOUNTING. With the introduction of channels and interrupts, the establishment of resident monitors or supervisors became an accepted fact. These supervisors were complex routines that could process interrupts, software requests, and a new language called the "command" language. Thus, the computer user could communicate with the system via com-

and language control cards that provided such information as name and account number, job limitations on time, pages, and cards to be used, and special resources (tapes, plotter, etc.) required.

Utilizing the hardware clock, most user interactions with the system were recorded, detailing what commands the system had received. The purpose of the accounting system was to monitor the individual user's interaction with the system and not imply the system performance.

Still, until the introduction of disk files, it was not feasible to verify each user's identification against some master file of valid users. Nor was it possible to determine user limits as to funds available, privileges, etc. Instead, accounting information was collected on magnetic tape or punched cards for later processing on an after-the-fact basis.

DISK FILES. The introduction of disk files added another step in automated accounting. Although not truly a random access storage device, a disk file was sufficiently fast to provide for an on-line verification of valid user identification. The accounting files could be permanent or resident files in the system, available only to the supervisor and the accounting system (i.e., privileged).

The accounting system could record each job transition or step, print out job charges at the completion of a job, and accumulate monthly statistics. By maintaining the accounting information in an on-line fashion, users could be prevented from using more than their current funds or exceeding their current account limits.

MULTIPROGRAMMING AND TIME SHARING. The more recent advances in computer hardware/software—namely, multiprogramming and time sharing—have produced the greatest impact on automated accounting. Because these advantages have made it feasible to allocate and share the multiple resources of a computing system, it is possible to have multiple users on the system at the same time. And as Nielsen (1968, p. 522) points out:

... in order to allocate the resources of the entire system so as to maximize the utilization of these resources, it is necessary to have not just an array of prices, but an array of flexible or adjustable prices that are responsive to demand. In other words, services as well as computer resources must be taken into consideration."

For each user of the system, the accounting system must know (1) who is responsible for the charges, (2) what type of service this user is entitled

to (and with what constraints), (3) what resources have been allocated to the user, and (4) what price schedule applies. Further, the pricing structure must allow the user to easily estimate and predict his costs, and should require only small amounts of system resources for the accounting.

UNBUNDLING AND PROPRIETARY SOFTWARE. With unbundling and time sharing has come new problems in automated accounting. Where it was previously possible to simply charge the proprietary system user for computer time used, more complex multiuser systems have made it possible to allow one user to provide service to another. The result is that users are billed by both the computing center (for hardware use, expendable supplies, etc.) and other users (for proprietary software use). Thus, the accounting system must be cognizant of the use of such proprietary software and should, in fact, allow some "higher-order" user to suballocate to another user some of the resources under his control. For instance, it should be possible for one user to develop and maintain a subsystem, fully consistent with the operating and accounting systems, which bills the individual users for actual resources used (both hardware and software).

Another example might be the course instructor who allocates fixed amounts of time or money to each student in his course in such a fashion that no student can use more than his share. Obviously, the person responsible for the account must be able to reallocate the resources without exceeding the total allotted to him. In addition, he should be able to place limits, which may not be uniform, on each student account so that special projects may use extra core or disk space, special hardware, etc.

Since some software can be charged only on a "value received" or transaction basis (such as ledger entries in an accounting system, or students scheduled in an automated scheduling system), the accounting system must be flexible in terms of the algorithm used to calculate actual charges.

Costs of an Automated Accounting System and Charges Levied.

The costs of an automated accounting system are directly a function of the resources used to gather and maintain the accounting information. In order that the overhead of collecting the information not interfere with normal system operation, the charges themselves must reflect the unique characteristics of the system. Normally, charges are based on such things as:

1. CPU time.
2. Memory time.

3. Connect time and/or port cost.
4. I/O operations performed.
5. Physical I/O units used:
 - (a) Cards read/punched.
 - (b) Lines printed.
 - (c) Magnetic tapes mounted.
 - (d) File space used.

However, these charges must relate to the characteristics of the operating system if they are to be easily collected. They should also relate to the allocation scheme for the resources if they are to be fairly levied (i.e., disk space should not be charged on a bit or character basis if it is allocated on a track or sector basis).

An on-line system where each user has an active account, although costly in terms of disk space required, allows the accounting system to—

1. Encumber funds on a per-job basis so as to prevent negative spending.
2. Set dynamic limits on controlled privileges as a function of time, geographic entry point, and system load.
3. Maintain flexible and dynamic pricing with actual cost information available to users on demand.
4. Maintain up-to-the-minute accounting for each user and periodically inform users of their accumulated computer resource utilization.

As described, automated systems can be fairly costly. However, the benefits provided both to the computing center operations staff (e.g., current resource use, system load, operating difficulties, etc.) and to the users (e.g., current pricing structure, resources available, job flow, etc.) generally outweigh the costs. Indeed, by knowing the state of the computing system, both operators and users are able to optimize their interaction with it so as to increase its effective utilization.

REFERENCES

1968. Nielsen, N. R. "Flexible Pricing: An Approach to the Allocation of Computer Resources," *Proceedings of the Spring Joint Computer Conference*, pp. 521-531.
1969. Rosin, R. F. "Supervisory and Monitor Systems," *Computing Surveys*, Vol. 1, pp. 37-54.

R. H. ECKHOUSE

COMPUTER-AIDED DESIGN

For articles on related subjects see **COMPUTER GRAPHICS; IMAGE AND PICTURE PROCESSING; INPUT-OUTPUT DEVICES; LIGHTPEN; TERMINALS; and TIME SHARING.**

Design is that creative activity which translates ideas into tangible reality. Engineering design has as its aim the production of specifications to allow manufacture to proceed and satisfactorily meet requirements.

Computer-aided design (CAD) concerns the utilization of computer systems for the purpose of design and communication of design information. The individual techniques used are not necessarily unique to CAD, but the particular combinations of computer system characteristics necessary for the full realization of CAD warrant special attention as a field of computer application. Design (and here we imply not just engineering design but any analogous activity that has as its objectives the output of manufacturing or construction information) may be considered on two quite distinct levels: first as a process of manipulating, analyzing, and assessing information (often of a geometric kind) in order to select options and optimize some performance characteristics against a given specification; and second, as a process of planning and information exchange between various organizations or parts of an organization in order to assemble and utilize the necessary design and manufacturing information in an optimum fashion. The first viewpoint emphasizes the special techniques of man/computer interaction that are sometimes taken to be synonymous with CAD', while the second viewpoint highlights the economically more significant factors of overall computer system design for effective information exchange.

Special-purpose analog computers were perhaps the first true design aids in that they provided models of systems, which could then be refined by experimentation, an analog not only of the system but of the mental processes of the designer. Design aids in the form of analysis programs were among the first uses of digital computers, and with the addition of output plotters they provided the ability to generate and visualize geometric information. The term computer-aided design became recognized following the work at M.I.T. in the late 1950s on the SAGE system and on multiaccess computing, which produced, respectively, the computer-driven cathode-ray tube display and the time-sharing computer systems that made interactive computing possible at

an economic cost.

The special power of interactive graphical displays lies in the iterative nature of design, normally a slow process bedeviled by lack of information. The interactive terminal permits many design options to be explored in a short time, and the addition of high-speed graphical visualization provides a means of data compression in a form familiar to the designer—a graph, diagram, or drawing. The light-pen as a means of input is also essentially a data compression device and is replaced as appropriate by keyboard commands, joystick movements, programmed function buttons, or stylus movements.

Time sharing has led to a massive growth in the use of multiaccess computers for both interactive and remote-batch processing in engineering, since it has made possible on-line working and interactive computing, if required, at reasonable cost. However, from the point of view of CAD, it is the multiaccess nature of the operating system and not the time sharing that is primarily significant, because it allows multiple access to files of information and so provides the essential basis for integrated information systems.

The multiple use of large computer systems made possible by time sharing in turn made feasible the development of advanced software, both at system and application levels. The linking of computers through data communications is a continuation of the trend that also favors the increased application of CAD through dedicated interactive graphics processors forming “terminals” in such a network. The two main aspects of CAD-interactive design optimization and large-scale information processing may thus be functionally separated while remaining part of an integrated operational system.

Special Equipment. There is on the market a wide range of printing terminals, of graphics terminals (particularly storage tube types), and of various sorts of add-on plotters to terminals attached to the ends of telephone lines. Some of these have been available for some time, and are generally well known. The ubiquitous Teletype hardly needs any description; in recent years a number of improvements have been made to this type of device, usually accompanied by a corresponding increase in price. A number of manufacturers have made available a plotting addition to the Teletype terminal. In most cases, this is driven by interposing a control box between the Teletype and the telephone line, so that a character stream from the computer is diverted to the plotter on appropriate occasions.

What has become popular in the last few years is the graphics terminal based on the storage-tube technique. Using a direct-view storage tube, a number of manufacturers have produced terminals on which may be displayed either alphanumeric or graphical information. At least one model is now available for less than \$5,000, and is capable of presenting a very clear line definition so that a complex picture may be painted on the screen. Storage tubes do have the disadvantage of not showing up particularly well in high ambient lighting conditions, and the erasure of chosen elements on the screen cannot be achieved except by clearing the screen and starting again. However, for certain applications they have proved to be more than adequate in real use and in many instances have become the equipment of choice. More recently, inexpensive terminals based on a conventional television display have made an appearance. Although for the present these are somewhat lacking in line definition, they can be seen in bright light, and the screen can be quite large.

An important development is the recent availability of the refreshed display terminal driven by a minicomputer, interfaced in turn to a communications line. These terminals are a development from the classical approach to computer graphics, except that they are very much cheaper, much more compact, and more restricted in the facilities they offer. The communications interface is asynchronous, so it looks to the host computer more like a conventional terminal. The absence of flashing lights is a relief, and the whole setup is small enough to stand on or sit next to a desk.

It is not within the scope of this article to discuss in any detail the pros and cons of any particular terminal. However, it is possible to point out one or two deficiencies in current ways of working. Much of the terminal equipment in use at the present time has been developed by specialist manufacturers who can be only partially concerned with the design problems to be solved. On the other hand, a clear understanding of what is needed to adopt CAD methods is still beyond the grasp of most computer system designers. It is hardly surprising that the exposure of available hardware to real design problems has shown some shortcomings in what is available.

An example is the simple storage-tube terminal used at the end of a telephone line. This is now commonly used for design work, fulfilling a dual role as an alphanumeric terminal and a graphic display. A difficulty that emerges when trying to use such a device in a real-life application is that the alpha-

COMPUTER-AIDED DESIGN

numeric information tends to get mixed up with the pictorial content, and a muddle ensues. The penalty incurred in avoiding the muddle is to clear the screen and to withstand another response-time delay. A frequently expressed desire is to be able to turn a hard copy device on and off, line by line, instead of having to deal with the contents of the whole screen. A solution to these problems seems to be in the development of a storage-tube terminal with an additional printing unit.

Finally, the use of powerful plotter systems should not be overlooked. These hardly come in the category of terminals for CAD, but do form an important ancillary to them. They can provide a very useful cover for the obvious deficiencies of a storage-tube terminal in the production of final accurate drawings, provided the graphics software facilities in the computer are correctly and rationally organized.

Computer System Aspects. Much of the early work in CAD was achieved by the use of dedicated computer power. Leaving aside for the moment the general-purpose computing done for engineering purposes, much of this early work was associated with graphic systems in which the display screen is directly refreshed from a stand-alone computer. The user sits in front of the display screen, which fulfills the function of an output device, while a variety of input devices may be available to the designer, such as lightpens, tracker balls, graphical tablets, and so forth. These systems can be highly interactive, giving the designer an almost instantaneous response to many of his demands. Much of the hardware has had an expensive air about it, and some of the display systems have been capable of manipulating pictures on the screen by means of hardware. Fundamental problems of picture manipulation include windowing and the rotation of three-dimensional wire diagrams. Windowing allows the user to display a small part of a large picture within finite screen bounds in two-dimensions, that is, in the X- and Y-dimensions, all parts of the picture outside these bounds being suppressed. Scissoring, or clipping, allows the user to control a three-dimensional display in the Z-dimension. Rotation, windowing, and even perspective projection have all been implemented as hardware options on some displays.

The designers' ability to input free-form information, as with Sutherland's Sketchpad and other similar devices, is an important element in computer-aided design. Three-dimensional flexibility is required, but this awaits a new breakthrough in equipment.

The disadvantage with the stand-alone graphics approach to CAD has been the expense. The capital cost of the equipment has been high and beyond the means of any but large organizations. Consequently, much of the pioneer work in CAD has been accomplished in university environments and in the larger companies. The advent of the simple storage tube meant that attempts had been made to run design processes on more conventional computer systems. In particular, the nature of the design program is that it has the characteristics of being rather large and yet spends a great deal of its time waiting for human intervention. On the face of it, it would appear to be a good candidate for running in a multiaccess system, and successful attempts have been made in this area. However, there remains the class of truly interactive design requirements that cannot successfully run in a multiaccess system, which is inevitably under management pressure to be optimized in revenue terms rather than in response terms. What may be required is a new design of multiaccess systems in which multiprocessor configurations are available to designers. In this way, each designer would have full private use of his individual processor whenever he is logged in. At the time of writing, attempts are being made to devise practicable systems along these lines, and it could be that this will be one of the major developments in the history of CAD,

Software Aspects. Perhaps the most important criterion in the design of software systems for CAD is portability. The cost of developing software tools and applications programs is often large in relation to the obvious economic benefits. In other words, the advantages of the CAD approach over manual methods are usually to be seen in terms of lead time to the finished product, rather than any direct reduction in costs. Consequently, it is of great importance that a software system developed on any particular type of hardware should be arranged in such a way that its use is not confined. Therefore most practical approaches in the applications area have been based on the use of **Fortran** as the prime language, although many software systems have been much more parochial in nature.

It is now of importance that software systems which have been in full and regular use in **general-** and special-purpose centers should be re-created in such a way that useful software facilities can be made available on computing systems other than those on which they have been developed. This requires a sensitivity by software writers to the need for using intermediate-level assembly systems and

languages, which have been little used so far despite many enthusiastic beginnings. One of the problems in this area is that the writing of graphics subsystems requires the use of a language that compiles very efficient code and yet which has a full range of arithmetic capabilities for matrix manipulation and the like. This dual requirement could account for the many individual and divisive approaches that have been made.

Applications. All CAD applications utilize to varying degrees the three special computer system capabilities of interaction, graphical visualization, and large-scale data communication. It can be argued that purely analytic or algorithmic design methods are possible and do not constitute CAD, but in practice it is generally found that such methods can be adopted only in restricted circumstances and that any attempt to produce a comprehensive design/manufacturing system leads to the adoption of CAD techniques.

The classic application areas of printed circuit board and integrated circuit design, structural design opti-

mization, and graphical part programming for numerical control machining were explored in an aerospace industry context in the 1960s and have now been highly developed. In such industries the complexity of design and the large work load have always put a premium on computer aids, and although the general approach adopted is relevant to other industries and applications, the specific solutions need to be modified to allow for different economic circumstances. In particular, it has not been found possible until recently to justify extensive use of dedicated graphic display processors although some pilot-scale installations have been used and have proved cost effective.

The developments that have recently taken place in types of display terminals, plotters, communication equipment, auxiliary storage (including cassette tape and disks), and in minicomputers means that, in contrast to the situation that obtained in the 1950s and 1960s, it is now possible to analyze an application requirement in detail and obtain computing facilities to suit, either from a specialized center or in house, or in combination. As a con-

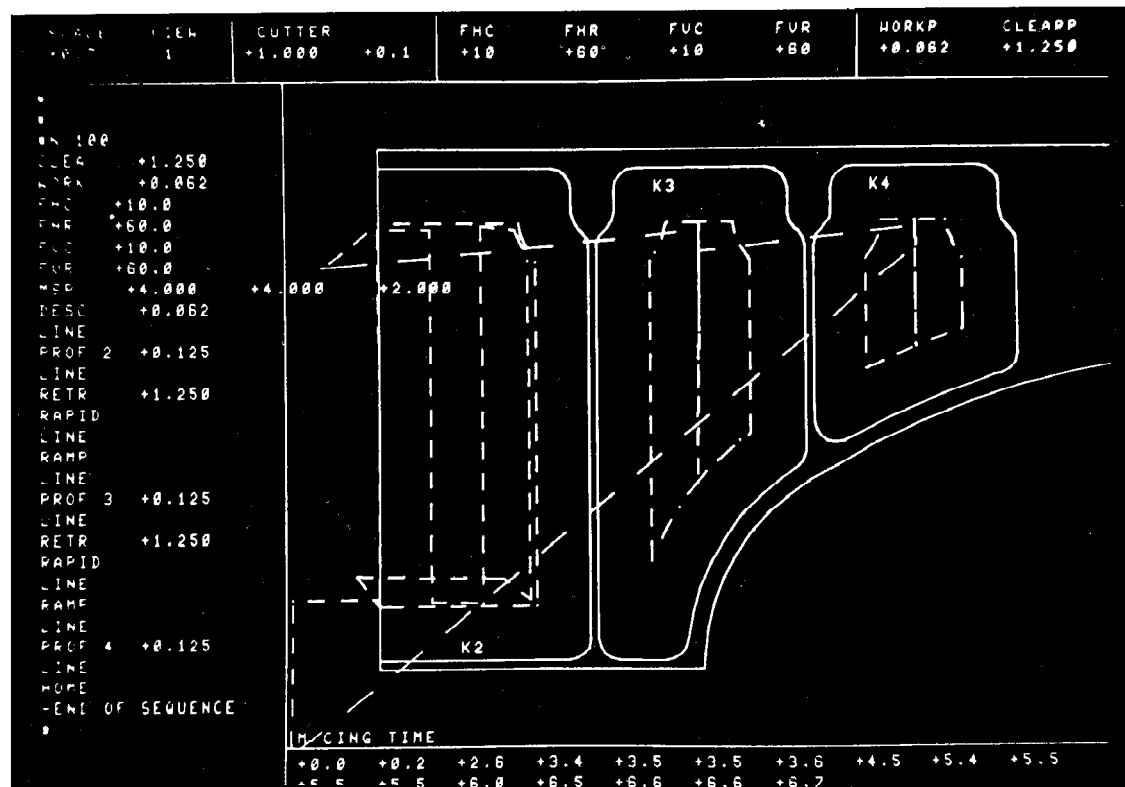


Fig. 1. Graphical numerical control programming taken from a storage-tube display.

COMPUTER-AIDED DESIGN

sequence, even classic applications like printed circuit board and graphical numerical control programming can be reoptimized in terms of cost, performance, and convenience. Fig. 1 illustrates graphical numerical control programming, using a low-cost storage tube terminal with a software system suitable for desk use on a free-standing mini-computer system. The same developments in time-shared computer systems and lower-cost equipment have made CAD more widely available in all fields of engineering and design.

The interactive use of computers through CAD is frequently aimed at design optimization. Optimum structural design becomes possible through on-line display of graphed output from analysis programs (Fig. 2). In the construction industry, increasing use is being made of computer-produced drawings for engineering purposes and halftone representations for environmental evaluation (Fig. 3). The design of buildings provides fertile ground for CAD, since design optimization to meet user requirements within the many constraints of site, cost, and materials is a complex process. Yet, with the development of coordinated dimensioning systems and component

standardization, it becomes possible to consider completely integrated "architectural" CAD systems. At the conceptual design stage, layouts and relationships may be explored quickly; during design realization, interactive methods permit optimization of structure and layout with visualization at each stage (Fig. 4); and in an integrated design system, data processing may be invoked on a large scale,

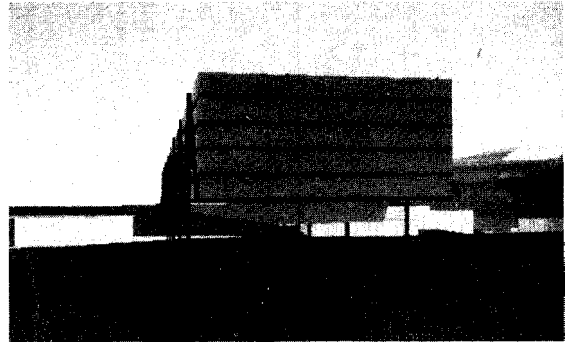


Fig. 3. Halftone representations for environmental evaluation taken from a refresh tube display.

DATA VALUES GO AGRAPH 1 SECTIONS 1

```

INPUT COMPLETE
*
CURRENT VALUES ARE
SPEED(RPM)=+1.75000000*3
FLOW(IGPM)=+254
HEAD(FT)=+56.5
Z AND Z1 =+1.00 +1.00
MIN. RAD.(MM)=+2.50
IMP. DIA.(MM)=+210
CUZ(M/SEC)=+10.3
BASE CIRCLE DIA(MM)=+225
R1MAX(MM)=+15.0
R2MAX(MM)=+15.0
ALFA1(DEG)=+15.0
ALFA2(DEG)=+15.0
WIDTH ON BASE CIRCLE(MM)=+30.0
ZZ =+1.00
ROUT1=+.000
OUTA=+.000
ZZZ=+.000
*
BRIEF VOLUTE DETAILS
THROAT AREA = 0.002708M**2
SEC#  ANGLE  HEIGHT  R1    R2    AREA
      DEG    MM      MM    MM    M**2
1      45     8.1    2.7    0.000254
8     360    39.8   11.6   11.6  0.001513
SEC#8 WIDTH= 47.0MM
SEC#8 THROAT AREA RATIO= 0.559
*
*
*12

```

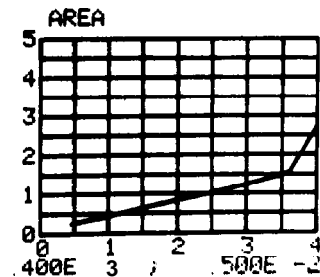
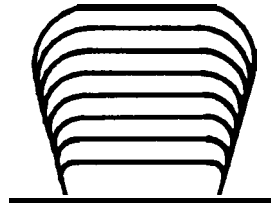
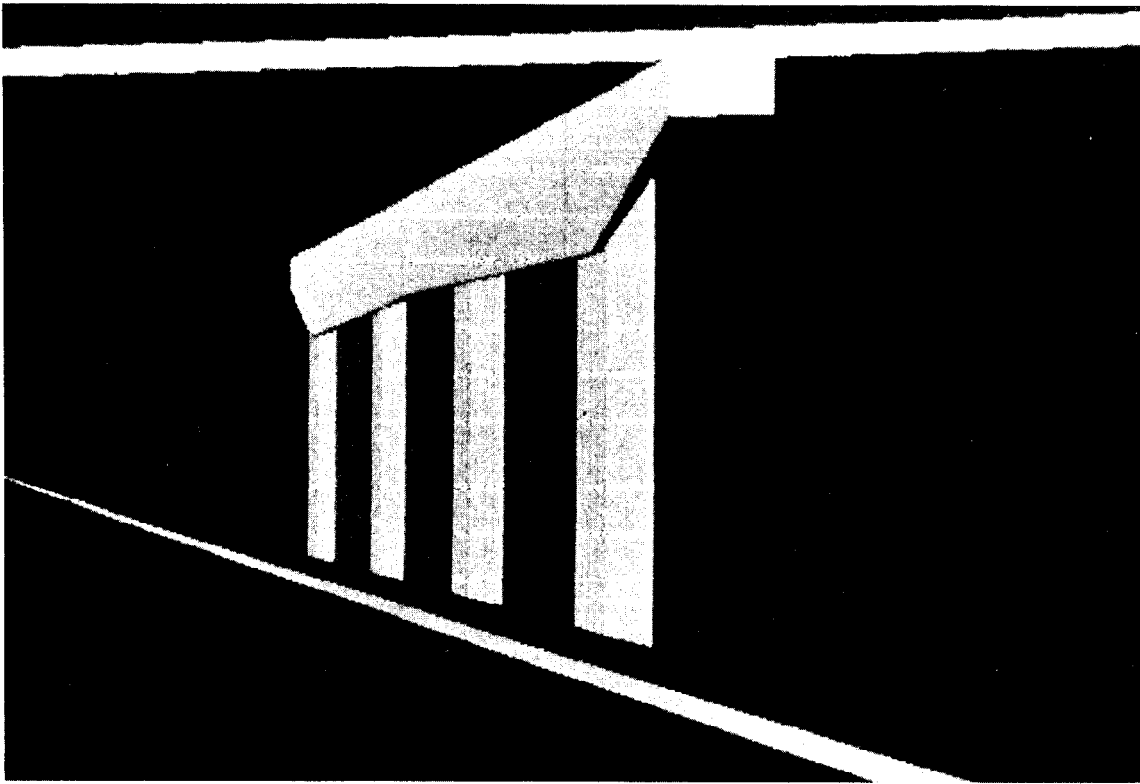
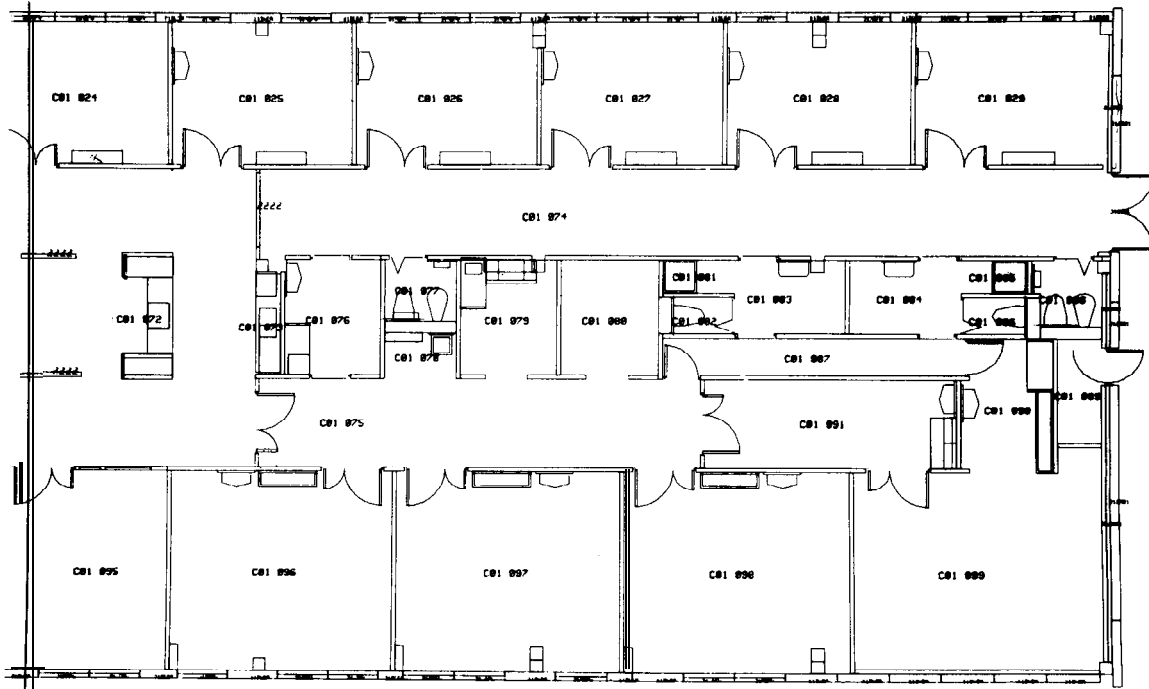


Fig. 2. On-line display of graphic output from analysis programs taken from a storage-tube display.



(a)



(b)

Fig. 4. (a) Structure; (b) composition showing architectural visualization of structure and layout.

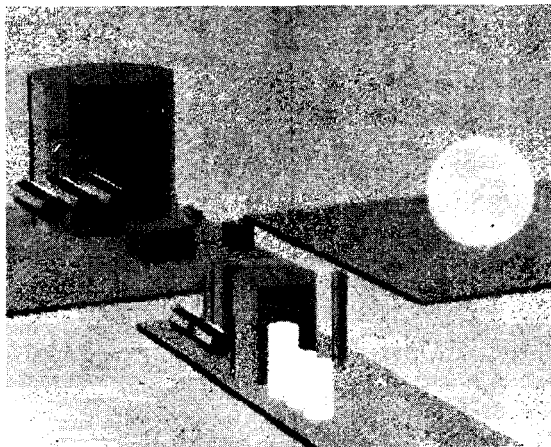


Fig. 5. Three-dimensional pipework layout in chemical plant design.

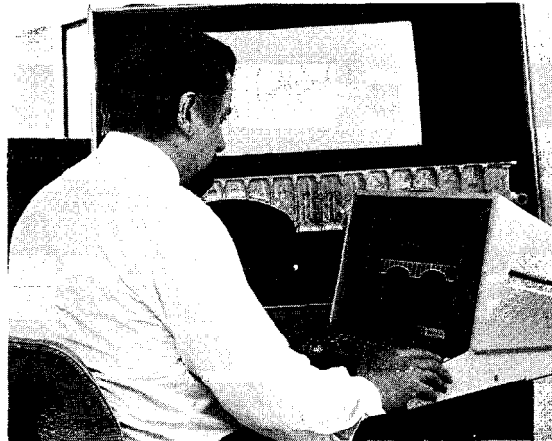


Fig. 6. Designs of engineering components.

both to utilize stored data on components and methods and to generate quantitative construction information.

Optimization of layout of components and their connections is a common requirement in circuit boards, architecture, and process plants. There are many related topological problems in these otherwise diverse activities, and this serves to illustrate one of the main connecting themes running through much of CAD—the use of descriptive data, structured in such a way that complex topological and geometrical relationships may be manipulated economically. Fig. 5 illustrates the semiautomatic optimization of a three-dimensional pipework layout in process plant design.

Mechanical engineering affords a wealth of CAD applications, even leaving aside the calculation-oriented activities of stress and thermal analysis, which benefit from interactive graphical methods of data preparation and results presentation. Problems may be classified as systems optimization, geometric description, or information systems. Programs for apparently simple tasks such as spring design must take into account the multiplicity of problem variables, and in these circumstances the interactive control of program data is advantageous. At the other end of the complexity scale, the optimization of internal-combustion engine layout at the preliminary design stage requires the manipulation of geometric as well as performance data, and interactive graphical methods may be considered essential to a full exploration of options in a reasonably short time.

Very many engineering components are two-dimensional in nature, and the ability to describe

and manipulate profiles via the display screen makes possible both the drafting and part programming of components (Fig. 6) and the computer-aided design of press tools. Three-dimensional geometry manipulation is among the more difficult design and manufacturing problems, and the use of either curve fitting or analytical surface geometry is finding a place in die and mold design, with benefits in the form of reduced lead time and increased precision (Fig. 7).

The esthetic aspects of three-dimensional components designed by such methods can be judged by perspective drawings and halftone (including stereoscopic) pictures, which make allowance for the solidity of the components, using data derived directly from the mathematical description used in the design techniques.

The real manifestation of CAD involves the integration of the design function within an information system embracing all functions of an organization. Such systems have come nearest to realization in computer manufacturing companies when the sheer volume of design data, test data, and modification data make computer-aided documentation essential. In mechanical engineering, progress toward integration is apparent in the batch manufacture of simple piece parts where CAD and numerical-control machining complement each other, and in the manufacture of standard ranges of equipment such as small heat exchangers or industrial gear boxes where rapid and accurate processing of documentation is a prime requirement.

The ultimate manifestation of CAD will carry this integration process a stage further and provide a system able to examine total design options, i.e.,

COMPUTER ARCHITECTURE

REFERENCES

1967. Gruenberger, F. (Ed.). *Computer Graphics - Utility-Product-Art*. Washington, D.C.: Thompson Book Co.
1970. Furman, T. T. (Ed.). *Uses of Computers in Engineering Design*. London: E U P.
1970. Wolfendale, E. C. A. *D. Techniques*. London: Butterworths.
1973. Hyman, A. *The Computer in Design*. London: Studio Vista.
1973. Vlietstra, J., and R. F. Wielinga (Eds.). *Computer-Aided Design*. Amsterdam: North Holland.

A. I. LLEWELYN

COMPUTER ARCHITECTURE

For articles on related subjects see ADDRESSING; ARITHMETIC-LOGIC UNIT; CHANNEL; DATA COMMUNICATIONS: INPUT-OUTPUT DEVICES; INTERRUPT; MEMORY: Auxiliary; MEMORY: Main; OPERATING SYSTEMS; SOFTWARE; and SUPER-C• MPU-TERS .

For articles on related terms see CACHE MEMORY; CONTENTION; CONTROL DATA CORPORATION 6000 SERIES; CPU; and VON NEUMANN MACHINE .

Computer architecture embraces the art and science of assembling logical elements into a computing device. As normally conceived, a computer architect accepts from a logical designer units such as adders, stacks, memory modules, and tape drives; puts them together so that they form a computer; and turns this over to a systems programmer, who then constructs an operating system for the machine. This should not be construed as being a passive role. The computer architect is responsible for (and, indeed, is almost the only one in a position to accomplish) the ideas interchanged between the two groups. He must bring software problems to the attention of the hardware types and hardware potentialities to the attention of the software types.

There are five fundamental components in any particular machine design. These are input/output, storage, communication, control, and processing. These are the five basic units of the simplest

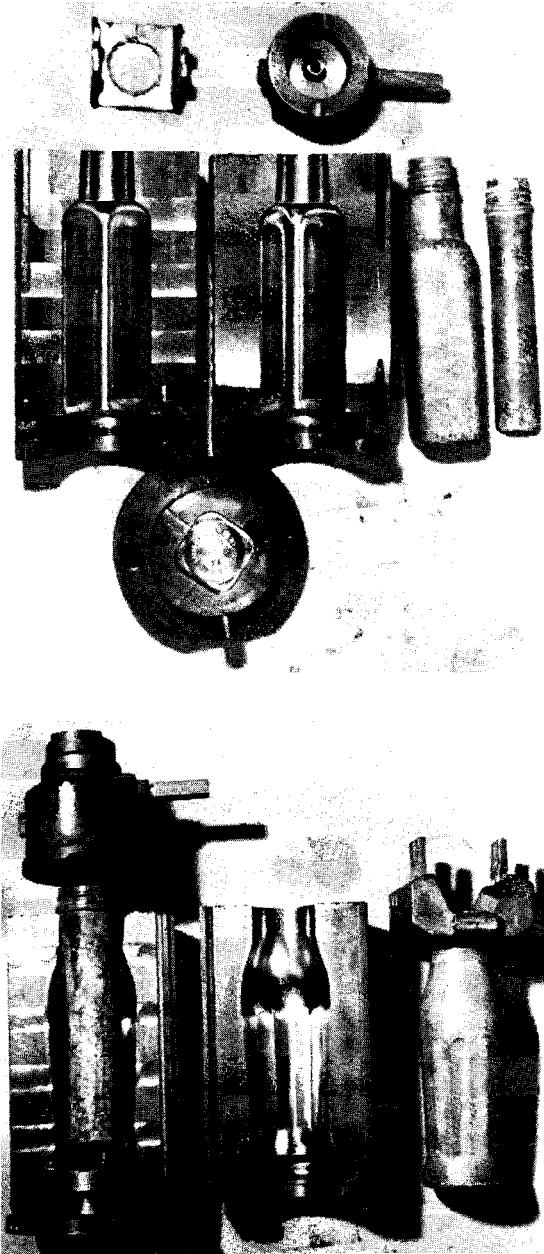


Fig. 7. Surface geometry in die and mold design.

trade-offs not just between design and manufacturing but also in service, use, and reliability. As technology progresses, this need will extend to an examination at the design stage of the balance of benefits to the community. The CAD system must therefore be arranged to take account of this to allow such participation.

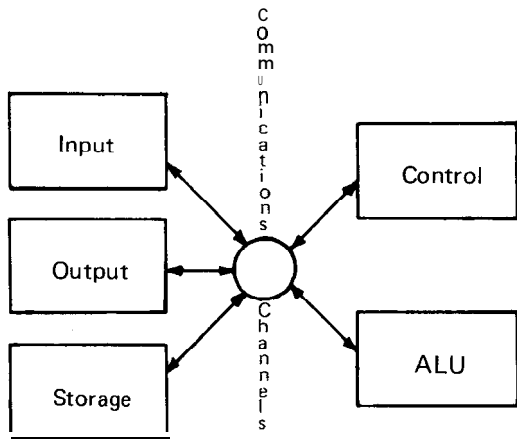


Fig. 1. Basic elements of every computer.

computer, as shown in Fig. 1. The devices on the left correspond to the "tape," the channel corresponds to the "head," and the units on the right correspond to the control mechanism of a Turing machine.

For reasons of cost and/or performance, various of these basic units are elaborated in different ways in extant machines. We will discuss the separate units and their complications one by one.

Input-Output. Although it is only through the input and output (I/O) devices that one can "converse" with a computer, surprisingly little attention has been paid to them architecturally. What has happened is that a separate channel has been provided between the I/O devices and the main storage device. Over this channel, information can flow into and out of the computer without tying up the arithmetic-logic unit (ALU) or the control unit. This requires that the storage units have more than one port (see below). The "channel" connecting I/O with storage has enough intelligence to be able to assemble and disassemble words, count how many units of information have been transferred, remember which I/O unit it is talking with, and where in main storage the information should go or come from; and finally, in some of the more sophisticated channels, it knows where to look to find more work to do when the present transfer is completed (chained I/O). Some of the larger machines may have several independent I/O channels capable of working simultaneously.

Storage. In addition to more or less continual technological advances, there are two basically architectural methods of improving storage device cost/performance.

The first of these has to do with overlap of references. In the earliest machines, and even today in the less expensive machines, only one storage reference could be going on at a time. That is, there was only one memory address register and one memory buffer register. In modern large machines, main storage is broken up into several separate blocks. Some subset of bits in the address are used to determine which block is being addressed, and the remaining bits are used to determine which word within the block is wanted. Since the blocks are independent, each with its own address and buffer register, it is possible to have as many storage references in progress as there are blocks. In theory then, for the price of a few extra registers, we can speed up storage access by a factor of N if there are N separate blocks of storage. In practice, of course, a program will often want a second thing from a given storage block before the first reference is completed. This is called "memory contention," and the more often it happens, the less benefit is obtained from dividing storage into blocks. Some contemporary computers use the high-order two or three or four bits of the address to select the block, some use the low-order bits (this is sometimes called "storage interlace"), and some actually use a couple of bits at each end of the address. All three methods have their proponents and disparagers.

The other basically architectural method of improving storage/cost performance can be given the general name of "paging." It involves a hierarchy of storage devices with two or more levels: one level fast but expensive; the other cheaper but slower. The idea is to get the data and instructions you are going to be wanting in the near future into the few fast words of storage you have purchased and keep the bulk of your program and data on the large, slow "backing store."⁷ If you are sufficiently clever about this, the user is deluded into believing that you have lots of "quite fast" storage for a price very near that of your competitor's slow storage. Perhaps the simplest example of this "paging" strategy is to fetch up several adjacent words from memory into a high-speed register every time the program references store. If successive instructions come from successive locations (as they usually do), or if successively required items of data come from adjacent locations, then one has the next instruction or datum right on tap in high-speed registers when needed and only, say, one reference out of four or eight (depending on how "wide" a fetch is made) needs to go down into main storage, while three out of four or seven out of eight run at a speed dictated by the register access time rather than by the main

store access time.

The next stage in this approach is to provide a moderately large number of high-speed storage locations. This has been called a "cache." On **FETCH** the CPU looks first in the cache for an operand or instruction, and only upon failure to find it there does it look in main memory (Fig. 2). Deposits (writes) are usually organized as a "write through" into main store so that main store always contains a valid up-to-date copy of all data. That is, information to be deposited is written into both the cache and the appropriate cell of main storage. Transfers of blocks of words between main store and the cache are performed strictly by hardware without software intervention. With appropriate fetching and assignment strategies, hit ratios (the probability of finding an item in the cache) as high as 0.8 have been observed.

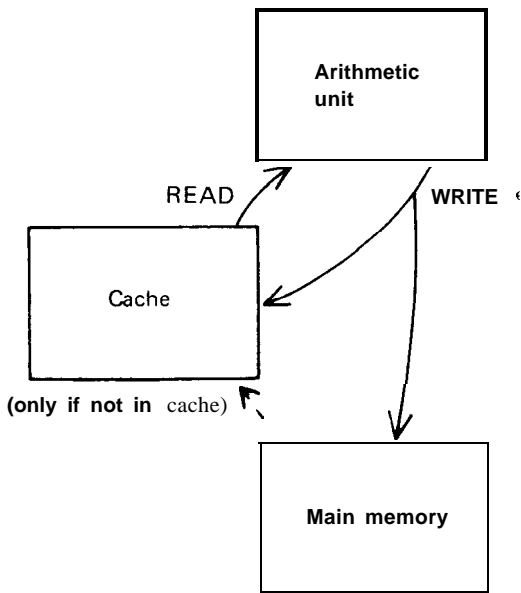


Fig. 2. Operation of reads and writes in a machine with cache memory.

The third stage in using a storage hierarchy is to swap programs and data between main store and a cyclically accessible device such as a disk or a drum. This is called "paging." Items are fetched up into main store as required, a page at a time. About 1,000 words seems to be a standard amount to transfer, although several other page sizes have been used or proposed. Since drum store is very much slower than most main stores, write-through is not done, and the current contents of an area of main storage must be

preserved (assuming they have been changed since coming up from drum) before new information is written into that area. The decision on which area of storage to overwrite is called the "page turning strategy" and is beyond the scope of this article. When a program references a nonresident datum or instruction, a "page fault" is said to occur and an interruption is raised, giving control of the processor to a portion of the operating system which then handles the matter via software. This technique of loading pages only as they are requested is called "demand paging." When loadings are specified in advance of reference to them, the technique is best called "overlaying."

Communication. Within a computer, data and information must be transferred from one place to another. In Fig. 1 the simple communication channel shown moves all information from one point to another under the direction of the control unit. In larger machines, communication becomes substantially more complicated.

We will define an "aggressive" device as one that can initiate a request for communication with another device; the device that honors that request we will call the "passive" device. If only the control unit can be aggressive, then it must periodically "poll" the input/output devices to see if any of them has anything to say. If I/O devices can also be aggressive, they may generate interrupts, which temporarily command the attention of the CPU in order to process the input or output and to ready further work for the device or channel.

Consider first the case where there is more than one peripheral device to be handled. Two principal modes of communication have been employed. The first of these is called a "radial selector" or a private-line arrangement.

In a private-line arrangement, each I/O device has a set of lines between it and the CPU for its own private use. The advantage of this scheme is that if a device is free, it is guaranteed that the pathway leading to it is also available. The disadvantage is that of high initial fixed cost for the decoding and driving of many sets of lines, even if only a few of them have peripherals attached.

Most contemporary machines use the other type of communication scheme called the party line, broadcast mode, daisy chain, or unibus (Fig. 3). The basic idea of this scheme is that one pathway connects the CPU to all peripheral devices. The active device (say, the CPU) broadcasts to all passive devices the name of the unit it wishes to communicate with. This unit, recognizing its own name, in

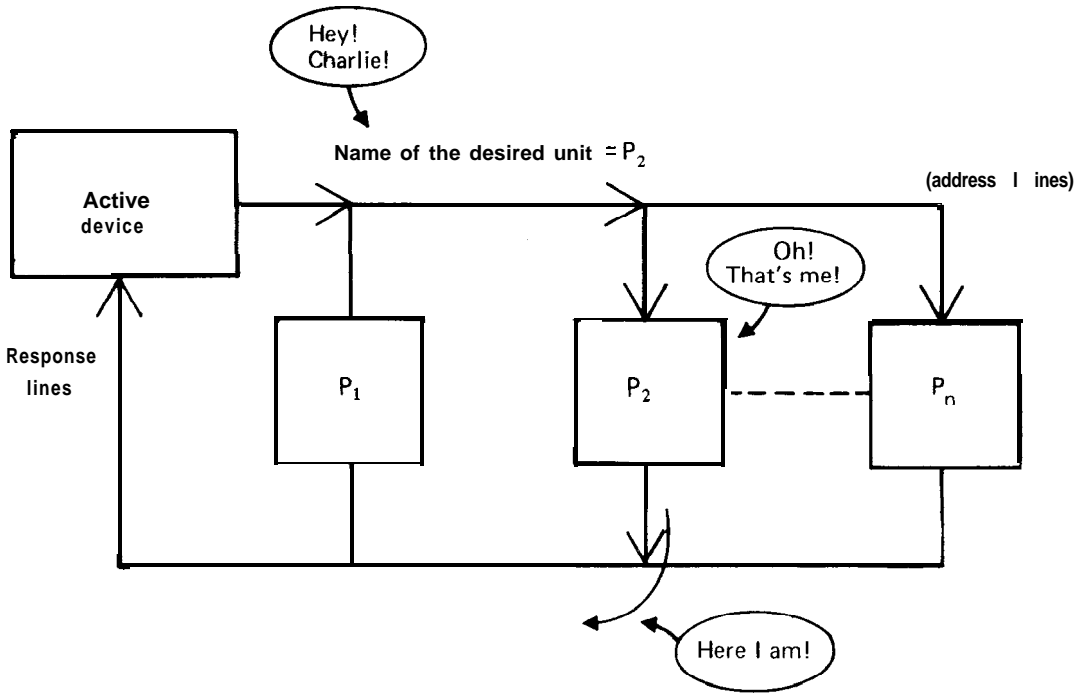


Fig. 3. Party-line selection.

effect holds up its hand, saying "Here!," and the dialog then ensues. The obvious problem here is that when one pair of devices is communicating, no other conversations can take place. Partial remedy may be had by keeping the occupancy of the "highway" short in duration. If this is not sufficient, more than one pathway may be installed, each called a "channel." Some devices may then be connected on each channel and some may be connected on more than one channel. For example, if several devices are connected to both of two channels, then any two of them may be active at a given time. The advantage of this approach is that only when a new device is attached to the computer need a "name recognize" be purchased,

When the attention of a device (CPU, memory, or even a channel) may be requested over several pathways, a "contention resolver" is required. That is, the device must give its attention to one pathway and (temporarily, at least) ignore all others. Such a resolver is often found in memory units that allow access from more than one CPU or from a CPU and an I/O channel. Sometimes this unit is called a "scanner." Two basic strategies are employed for the resolution of contention. The first is called "round robin." Suppose there are three pathways, A, B, and C, connected to a memory (Fig. 4). A is examined to

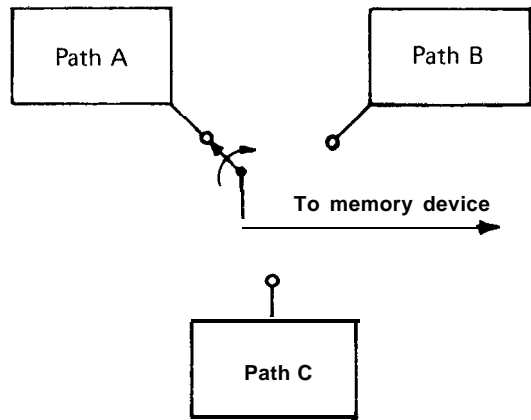


Fig. 4. A round-robin type of contention resolver. Service is offered to paths A,B,C,A,B,C,A, etc.

see if it needs service. If it does, it receives it. If it does not, or when the service is completed, the scanner examines B. Then it examines C, and then back to A and around the circle once again. If there are N pathways, each one is guaranteed at least $(1/N)$ th of the memory cycles. No pathway gets

better service than any other and all are treated equally.

The other basic scanning method is called the "resetting scanner." After servicing a pathway, the scanner looks at the list, starting over again from the beginning. Only if pathways *A* and *B* are both idle will *C* be able to get into memory. This gives pathways early in the scanning sequence priority over those later in the sequence. Such priority is extremely useful for impatient devices such as drums or tapes, where data may be lost forever; at least, it may cause severe inconvenience if the device does not get attention within a short time after demanding it.

Control. The control unit of a computer is charged with fetching and decoding instructions and then computing the addresses of required operands and fetching them. Finally it must select which portion of the ALU will be required to perform the instructions and transmit the required data to it. In order to speed up this process, the technique of "instruction lookahead" has been employed. To begin with, the control unit is designed to "fetch ahead"—i.e., whenever the path to memory is available and there are less than *n* unexecuted instructions on hand: Go get some more. This may be accomplished by (1) packing two or four instructions into each word and fetching up single words; (2) fetching up four words at a time, each with one or more instructions; (3) keeping a file of the eight most recent instruction word fetches and going there for the instruction in preference to main memory. Once more than one as-yet-unexecuted instruction is on hand (in the control unit), it becomes possible to start working on them before their "turn" actually comes up. (The obvious danger, of course, is that because of branching their turn may never come up.) We fetch instruction *i* while we are decoding *i* - 1, while we are simultaneously (in independent hardware) computing the effective addresses of *i* - 2 while we are actually executing *i* - 3.

Provided the execution of none of these instructions modifies other instructions in the stream (this is reasonable for reentrant coding) and provided none of the instructions references registers that other instructions are in the process of changing (this is somewhat less safe), and provided none of the instructions in the stream changes the flow of control (i.e., are branches)—and this last is ridiculous; approximately one out of every four or five instructions executed has been found to be a branch—then instruction lookahead is worthwhile, but even if some of or all these provisos are not satisfied,

instruction lookahead may still be useful.

One way to avoid the problems of interacting instructions described above and to realize the potential of instruction lookahead is to take successive instructions from different, independent programs. This is perhaps the ultimate in multiprogramming in which one instruction from a program stream gets executed before moving on to the next program. The Honeywell 800 and the peripheral processors of the Control Data CDC-6600 do exactly this. This approach does not help get a particular job done quicker; it "merely" increases the throughput of the computer by having several jobs running at once.

Arithmetic-Logic Units. The part of the machine in which arithmetic and logical operations are actually carried out is called the ALU. Two schemes for speeding up an ALU have been used. Both depend on instruction lookahead to provide several instructions to work on at the same time, and consequently both suffer degraded performance when lookahead does. These methods are called "pipelining" and "division of labor." In both methods we try to increase the number of instructions being completed per second by increasing the number going on simultaneously. Pipelining may be likened to an assembly line, whereas "division of labor" might be thought to be similar to a job shop operation. In the first type, a part of the hardware unit does a little bit for instruction *i* + 1 just as an automobile assembly line puts an automobile together a bit at a time. Sometimes a pipeline may have as many as 20 or 30 stages and be capable of processing that many separate instructions. One instruction may take half a microsecond to pass *through* the pipeline, but the pipeline can accept another new instruction every 25 ns.

The other method of speeding up an ALU is to design specialized units dedicated to performing particular tasks rather than having general-purpose units. Thus, we may find a boolean unit, a floating-point multiplier and divider, a floating-point add/subtract unit, a fixed-point arithmetic unit, and others. These are sometimes called "functional" units. The IBM 370/195 and the CDC CYBER 73 both have multiple functional units. Of course a designer may combine pipelining with multiple units and have several pipelines—either a general-purpose type as does the CDC-STAR or a special-purpose type.

Several recent studies have shown that even when given as much hardware as might be desired, the interaction of instruction streams and the inter-

COMPUTER-ASSISTED INSTRUCTION (CAI)

dependency of successive data references limit the average number of instructions that may be executed simultaneously (from a single program) to somewhat less than two (Riseman and Foster, 1972).

Other Designs. Several other ways of designing computers exist, besides the way described above. The best known such design is the "array" type of computer and the earliest example of this approach is the machine of Unger (1958). Two contemporary examples of array-type machines are the ILLIAC IV built at the University of Illinois and the STARAN computer built by Goodyear Aerospace Corporation. An array machine is characterized as one having many pieces of data being simultaneously processed by one program. STARAN has the further interesting property that its memory is content-addressable.

The other major variation in computer design is due to von Neumann and is called a "tessellated automaton" or a Holland machine, after John Holland who described a definitive version of the type. In this type of machine an unbounded, usually planar, array of cells each hold (and are capable of executing) an instruction. A program is a collection of physically adjacent cells, with one or more of them "active," i.e., currently executing its instruction. Holland's contribution was to show that arbitrarily many programs could be simultaneously active in such a machine. Von Neumann showed that a program could be written which reproduced itself. No physical embodiments of tessellated machines of any appreciable size (more than 100 cells) have yet been constructed, nor are there present plans to build any.

A computer with more than one processor is some times called a "multiprocessor." Depending on how the work load and job functions are divided among the several processors, one can have a master/slave configuration, a front-end/back-end configuration, a "first among equals" organization, or an each-man-for-himself type of chaos.

When the number of connected processors exceeds some small integer, people begin to speak of parallel processing. Clearly, ILLIAC IV and STARAN are of this type. Not nearly so clear is whether a machine like a CDC 6500 with two central processors and ten peripheral processors should be called a parallel processor. Usually it is not, but it does meet the normal criteria.

REFERENCES

1958. Unger, S. H. "A Computer Oriented Toward

Spatial Problems," *Proc. IRE* (October), pp. 1744-1750.

1970. Foster, Caxton C. *Computer Architecture*. New York: Van Nostrand-Reinhold-

1971. Bell, C, Gordon, and Allen Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill.

1972. Riseman, E., and C. Foster. "The Inhibition of Potential Parallelism by Conditional Jumps," *IEEE Trans. on Electronic Computers*, Vol. C-21, No. 12 (December), pp. 1405-1411.

C. C. FOSTER

COMPUTER-ASSISTED INSTRUCTION (CAI)

For articles on related subjects see **COMPUTER-ASSISTED LEARNING AND TEACHING**; and **COMPUTER-MANAGED INSTRUCTION**.

Computer-assisted instruction (CAI) refers to the use of computers to present drills, practice exercises, and tutorial sequences to the student, and perhaps to engage the student in a dialog about the substance of the instruction. A CAI (tutorial) dialog is achieved between computer program and student when the responses derived from the program are highly responsive to the questions, answers, and directives given by the student, while at the same time the dialog advances the goals and means established by the author of the curriculum materials.

CAI is only one part of computer assistance in the processes of learning and teaching. It has proved successful where the goals of instruction are clearly defined, achievement of those goals is highly valued by the organization providing instruction, the substance of instruction is suited to automated delivery, and the student is lacking important skills, background, or motivation for self-instruction via less expensive media. Research studies tend to show advantages for CAI in terms of shorter learning times and improved performance. Inhibitors to operational use include high costs of delivery systems and curriculum development, conflicts between individualized instruction and current educational practices, and commitment of most of the computing resources available in schools to instructional use for education about computers. A comprehensive analysis of CAI effectiveness, along with other media for

instruction, is given by Jamison, Suppes, and Wells (1974).

Major Demonstration Projects. A group of engineers and educators in the Computer-based Education Research Laboratory at the University of Illinois, Urbana, have designed a computing system (PLATO IV) especially for effective and efficient teaching. It is a large system intended to provide instructional computing to over 1,000 simultaneous users throughout the University and also a number of other colleges and schools in Illinois. The design includes notable advances in the technology for display and special arrangements for efficient communications through broadband channels of the telephone system. University faculty are authoring computer-based curriculum materials for many subject areas and at various levels of education.

Stanford University has operated a CAI system

that distributes instructional computing to a number of centers throughout the country. It is a large-scale service operation using long-distance telephone communications, clusters of terminals, and some stand-alone computer systems. The remote centers are usually associated with elementary school demonstration projects and special education institutions. The service operation is conducted in parallel with an extensive program of research and development at the Institute for Mathematical Studies in the Social Sciences, Stanford University. Curriculum materials have been prepared for young children (elementary school math and reading), learners with special difficulties (for example, the deaf), and certain university courses (especially second-language learning and logic).

TICCIT (Time-shared, Interactive, Computer-Controlled Informational Television) is a name given to systems being developed by the Mitre Corpora-

James Smith was found unconscious under suspicious circumstances. You are the Scotland Yard Inspector. Proceed with your investigation.

WHO ARE THE SUSPECTS?

The wife, brother and partner presently are **considered** the suspects.

WHAT IS THE MEDICAL REPORT?

The victim is in a coma. He has wounds on his head and symptoms of poisoning are evident,

GIVE ME THE CONSTABLE'S REPORT.

A glass was found next to the victim's bed, and a bottle labeled poison in the closet. The window has a bullet hole in it. A bloody pipe wrench was found in the shed.

I WANT TO INTERROGATE THE WIFE.

I advise you request the lab tests first.

LAB TESTS

This is the lab.
What is it you want?

WHAT DO YOU FIND ON THE GLASS?

This is the lab.
Fingerprints found which match those of the wife.

MEDICAL REPORT

No change in the condition of the victim.

ANY NEW SUSPECTS?

No new suspects.

The program recognizes "suspects" and provides the stored response.

"medical" is recognized.

"constable" is recognized.

The program recognizes "interrogate" but will not permit exploration of that branch until another one has been fully explored,

After the wife has **been interrogated** this branch of the program will release more incriminating information.

"medical" is recognized a second time.

The program includes a "stack" of replies for some directives so that on subsequent requests for **information** a fresh reply is given.

Fig. 1. Sample of CAI learning exercise (exerpted from Bolt, Beranek, and Newman program).

COMPUTER-ASSISTED LEARNING AND TEACHING

tion in McLean, Virginia. The first version of an instructional system is designed especially for use in a small college. It is based on a medium-sized computer system and video technology to obtain low-cost operation with about 100 simultaneous users. The hardware and software design has been coordinated with the development of instructional materials, which are very carefully prepared according to certain rules of effective instruction. The first set of materials is being prepared by instructional design teams at Brigham Young University in Utah to provide basic remedial instruction in mathematical and language skills at small colleges.

The list of schools and colleges pursuing interesting development programs is very long. One of the long-standing operations combining development and applications is in the Philadelphia public schools. Activities there, under the direction of Sylvia Chorp, include a wide range of CAI along with many other uses of computing in the instructional program.

Sample of Tutorial Dialog. Fig. 1 is an example of CAI taken from a student learning exercise in gathering information and making decisions. As a demonstration, it requires no specific content knowledge; the application of this computer teaching strategy is common in physics, chemistry, biology, and medical diagnosis where knowledge of the subject is essential to success in the exercise. The computer program guides the student in the exploration of a tree of possibilities, some of which lead to the solution of a simulated problem through reports, laboratory tests, and direct interrogation.

Areas of Application. CAI materials have been prepared for many subjects, from accounting to zoology, and from preschool through adult education. One representation of the curriculum datum base is provided by Hoyer and Wang (1973). Many more materials can be found in selected discipline areas by consulting teaching publications or professional committees associated with mathematics, physics, chemistry, biology, geography, political science, history, business, engineering, law, and medicine, among others.

REFERENCES

1973. Hoyer, Robert E., and Anastasia C. Wang (Eds.). *Index to Computer Based Learning*. Englewood Cliffs, N. J. : Educational Technology Publications.

1974. Jamison, Dean, Patrick Suppes, and Stuart Wells. "The Effectiveness of Alternative Instructional Media: A Survey," *Review of Educational Research* (AERA), Vol. 44, No. 1, pp. 1-67.

K. L. ZINN

COMPUTER-ASSISTED LEARNING AND TEACHING

For articles on related subjects see **AUTHORING LANGUAGES AND SYSTEMS; COMPUTER-ASSISTED INSTRUCTION; COMPUTER-MANAGED INSTRUCTION; and NETWORKS FOR INSTRUCTION.**

For articles on related terms see **EXTENSIBLE LANGUAGES; INFORMATION PROCESSING; SIMULATION; and SPEECH RECOGNITION.**

The impact of computers on teaching and learning activities at all levels of education is considerable and the extent of use is increasing rapidly. Current uses in post-secondary education are quite varied. A medical student practices diagnosis and prescription on a wide variety of hypothetical patients simulated by computer programs. A senior engineering student using computer assistance solves problems in road design which ten years ago were not approached until after two years of experience on the job. A sophomore in computer science develops a program to help his professor of chemistry evaluate the effectiveness of questions on a multiple-choice quiz. A freshman in general psychology directs a computer-based information system to assemble a complete bibliography on the relation between achievement motivation and college grades, which is as current as the journals received by his professor. A laboratory technician tests himself on newly acquired skills, using a terminal on a hospital information system.

Computing is also quite visible in education outside colleges and universities. A high school science student applies wildlife management practices to a computer simulation of the American bison herds that were slaughtered in the 1800s. An English literature student programs a computer to generate poetry. A child in the fifth grade explores mathematics by writing computer programs that draw

COMPUTER-ASSISTED LEARNING AND TEACHING

spirals or solve mazes. A bedridden second grader practices addition problems "spoken" by a computer over the telephone; the computer checks the answers that the student enters on the Touchtone telephone buttons; or, in some experimental systems, the student simply speaks the answer into the telephone to be "recognized" by the computer. A high school dropout improves language skills using a computer program made available on a community television cable system.

When the computer system is appropriate for educational uses and the programs are properly written, the learner should find the assistance to be: responsive to his needs; patient and not punitive while he learns; accurate in assessment of answers and problem solutions; individualized in a useful way; realistic in the presentation of training or testing situations; and helpful with many information processing tasks. Teachers find computer assis-

tance valuable for keeping accurate records, summarizing data, projecting student-learning difficulties, assembling individualized tests, and retrieving information about films or other learning resources. Authors of textbooks and other learning materials use computers to draw figures, to animate motion picture sequences, or to keep track of the introduction and frequency of occurrence of concepts throughout a text. Researchers record and analyze data, build models of student learning and performance, and administer experiments on methods of instruction. Administrators use computers for keeping records, planning, scheduling, allocating resources, and processing data.

These applications and many others are described in a rapidly growing literature. The single most comprehensive publication is the proceedings of an international school (Zinn et al., 1973), which also includes a list of many other sources, both

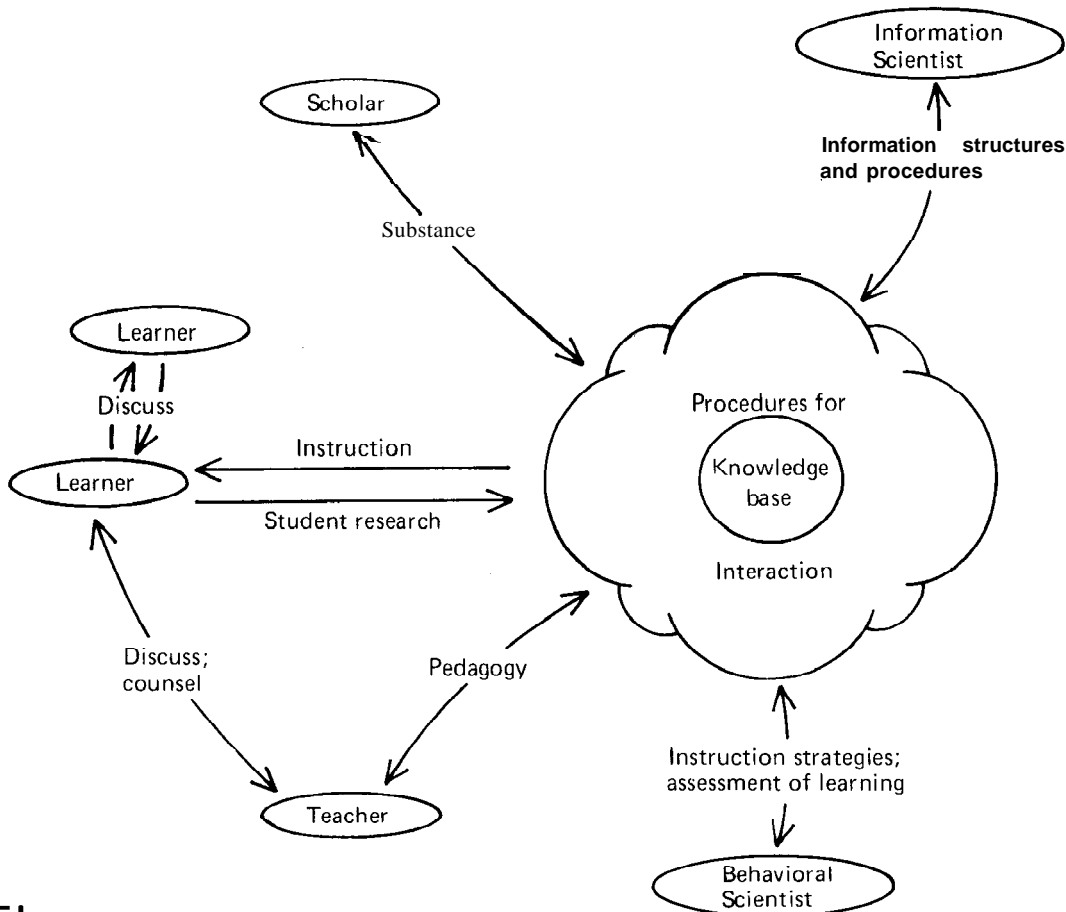


Fig. 1. Human components in effective computer use for learning and teaching

COMPUTER-ASSISTED LEARNING AND TEACHING

introductory and highly technical. An important guide for computer use in higher education is the report of a RAND Corporation study for the Carnegie Commission on Higher Education (Levien, 1972). The *Annual Review of Information Science and Technology* carries a scholarly review every three to five years (Silberman and Filep, 1968; Vinsonhaler and Moon, 1973).

Use of the computer as a tool for problem solving in education began in graduate schools about 1955, and a few years later moved into the classroom with the initiation of curriculum development projects in engineering and sciences. Computer use as a teaching machine dates from 1958; early developments took place at IBM's Watson Research Center, System Development Corporation, and the University of Illinois Coordinated Science Laboratory. The topic of computers in education became popular for meetings in 1965; separate conferences were held on computers in American education, higher education and physics teaching. In the following years of rapid growth, major conferences were organized for computers in mathematics teaching, chemistry education, computer science education, science education, undergraduate curriculum, and high school counseling. Instructional use of computers is a frequent topic at meetings of the contributing professions (computing, engineering, psychology, and educational research) and at meetings of teachers of most disciplines (ranging from engineering and physics to history, art, and modern languages). Various human components in effective computer use are related in Fig. 1.

A special-interest committee on computer-assisted instruction was formed within the Association for Computing Machinery in 1967 and reorganized as the Special Interest Group on Computer Uses in Education (SIGCUE) in 1969. The *SIGCUE* Bulletin is a good source of current information. The Association for Development of Computer-Based Instructional Systems (ADCIS), derived from an IBM instructional system users group formed in 1966, distributes an informative newsletter. Many journals carry reports of research and development for instructional uses; in particular: *Instructional Science*, *International Journal of Man-Machine Studies*, *Journal of Educational Technology Systems*, and the *IEEE Transactions on Systems, Man and Cybernetics*. Magazines regularly including information are *Educational Technology*, *Computers and Automation*, *A EDS Monitor*, *Datamation*, and *Computer Decisions*. In addition to the newsletters and bulletins of professional groups, vendors distribute periodicals on educational uses, e.g., Digital Equipment Corporation,

Hewlett-Packard Company, and Wang Laboratories.

Kind of Use. Computer assistance with learning and teaching has been described by many different phrases. One could follow the word "computer" with two terms, one from each of the following lists :

- | | |
|------------|-------------|
| -aided | training |
| -assisted | instruction |
| -augmented | learning |
| -based | teaching |
| -extended | education |
| -managed | |
| -mediated | |
| -monitored | |
| -related | |
| uses in | |

The most common label 'has been **CAI**: computer-aided instruction. When "instruction" is replaced by "learning", as in **CAL**, the combination connotes greater emphasis on activities initiated by the learner than on the instructional materials created by a teacher-author. When "learning" is replaced by "education" to obtain **CAE** (or **CBE**, computer-based education), the implication is a greater variety of computer uses, including administrative data processing and materials production as well as student use of computers. If the role of the computer is to assist the teacher in managing instruction, for example, in retrieving and summarizing performance records and curriculum files, the label used is **CMI** : computer-managed instruction.

INSTRUCTION AND THE LEARNING PROCESS. The most visible use of computers in instruction is to provide direct assistance to learners and to assist teachers, administrators, and educational technologists in helping learners. The users may work individually or in groups, using a device directly connected to a computer (on line) or using some medium later entered into a computer (off line), typing letters and numbers only (alphanumeric) or pointing and drawing diagrams for the computer (graphic), and so on through many options that vary in cost and convenience. Some typical labels within this category of use are: drill, skills practice, programmed tutorial, testing and diagnosis, dialog tutorial, simulation, gaming, information retrieval and processing, computation, problem solving, construction of procedures as models, and display of graphic constructions. A very popular use of the computer is for simulation of a decision-making situation, as in resource management, pollution control, business

COMPUTER-ASSISTED LEARNING AND TEACHING

marketing, or medical testing. For example, college economics students study the history of a hypothetical national economy (similar to that for the United States), prescribe actions such as changing the prime interest rate, and observe the consequences for unemployment, inflation, and other indicators. Time is much compressed in the hypothetical situation, and real-world complexities are abstracted for easier study.

MANAGEMENT OF INSTRUCTION RESOURCES AND PROCESS. Computer aids help teachers to supervise the instructional process, and similar assistance is provided directly to students without intervention of teachers and managers. Information management services are readily extended to potential users of learning resources outside traditional educational institutions. The essential information in the various files for management of instructional resources concerns student performance, learning materials, desired outcomes, job opportunities, and student interests. For example, a student obtains information from the computer about his achievement and then compares his own performance, interests, and goals with averages recorded for all similar students using the information system. After interpreting the information provided, the student uses the computer further to locate and retrieve suitable learning aids from a large file keyed to goals, learning difficulties, job opportunities, and interests.

PREPARATION AND DISPLAY OF MATERIALS. Materials may be generated in "real time," i.e., as needed by a student in a seminar or by a teacher during a lecture. Text and problems also may be assembled by computer in advance of scheduled use so that individualized materials may be distributed at less expense than through on-line computing. Computers assist writers of materials in many ways—for example: procedures for generating films and graphs; on-line trial of materials under development; procedures for automatically editing and analyzing text materials for new uses, and information structures for representing new organizations of knowledge; hierarchies of instructional objectives; and libraries of learning materials. New technologies are changing the work of technicians and teachers in developing educational materials and media. Machines handle much of the routine in drafting graphics and editing film.

OTHER USES OF INFORMATION PROCESSING. Those planning instructional uses also attend to the educational implications of computers in administration (accounting, scheduling, planning, etc.) and in research (institutional, sociological, psychological, instructional, etc.), and to the practice of various

computer-related vocations in science, technology, management, banking, production, retailing, etc. The last area is especially important because of needs for preservice training. For example, most large retailing operations use computing heavily in many areas, and employees with some sensible background in computing have a better chance of coming to terms with computer assistance on the job. Indeed, a general literacy about computing and information processing may be essential in the age of informatics. Educated persons should have sufficient knowledge about the practices of automated information processing to exercise on occasion effective control over the machines and data files with which they must deal.

Means And Goals

DIVERSITY OF RESOURCES. Many different kinds of computer and software systems are being used by research and development projects today; a few are being marketed for operational use. Some small machines can be used by one or a few students (Fig. 2) at a time to access stored programs (usually drills or simulations) or to write simple computer programs. Slightly larger systems dedicated in a similar way to interactive instruction have been programmed for simultaneous use by as few as 4 and as many as 40 students. Larger systems in operation now can handle up to 200 students accessing a variety of programs. The PLATO system at the University of Illinois has been planned for up to



Fig. 2. Two students study science with aid of a computer in Project LOCAL, one of the first to bring computing into the schools for student use.

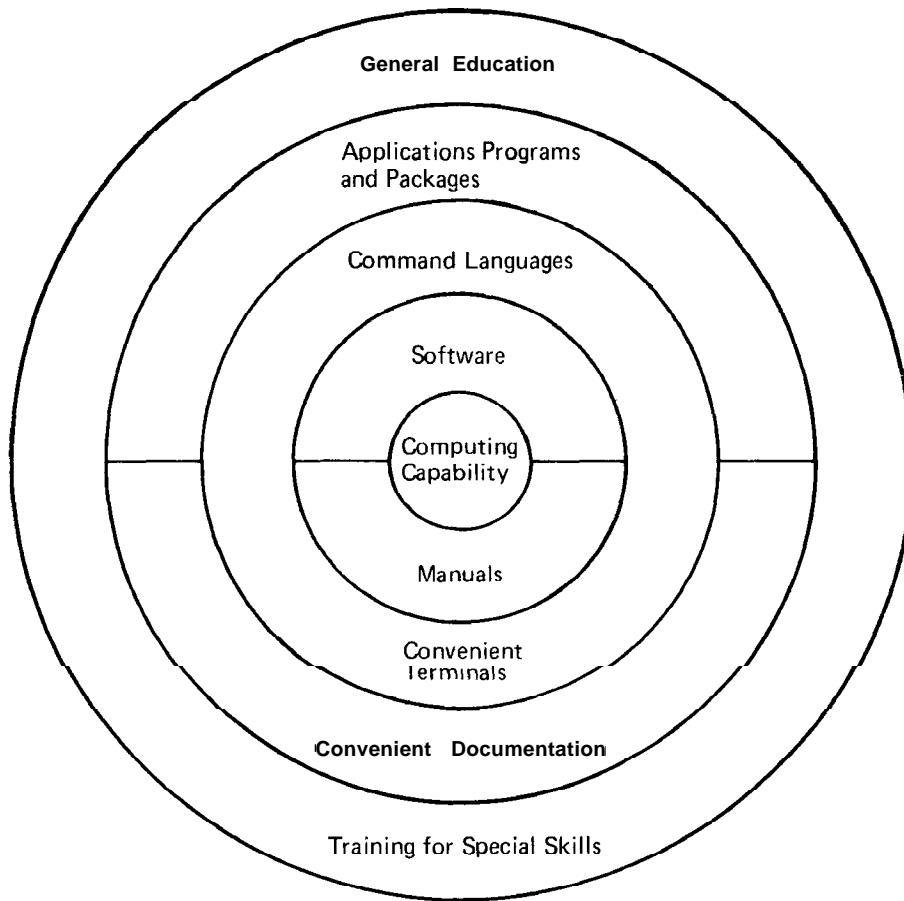


Fig. 3. A perspective on software and services for users.

4,000 simultaneous users and diverse applications: self-instruction, self-testing, simulation, gaming, and problem solving. Many of the multipurpose computer systems serving general user communities at colleges and universities include instructional applications among other uses for research and administration. Examples can be found at Dartmouth College, Massachusetts Institute of Technology, Carnegie-Mellon University, the University of Michigan, and Stanford University, to name only a few.

Programming languages and systems (software) exhibit even more diversity than the computing equipment (hardware). More than 40 languages and dialects have been developed specifically for programming conversational instruction, although many programs have been written in general-purpose languages such as **Fortran**, **PL/I**, and **Basic**. Different kinds of users have distinguishable requirements: students, instructors, authors, instructional researchers, administrators, and computer

programmers working on convenience programs for any of the other users. The characteristics of different subject areas also necessitate different language features. Authoring languages and systems are described in a separate article. Fig. 3 summarizes a perspective on software and services for users; a set of concentric circles represents successive levels of access for users approaching the core of computing capability. Fig. 4 represents the perspective of the user at the center.

Instructional materials (sometimes called "courseware") have been written in almost all subject areas and for many age levels, including pre-school reading, elementary school science laboratory, intermediate school social studies games, high school biology laboratory, college mathematics, introductory German, physics, chemistry laboratory, and professional school exercises in business management, medical diagnosis, architectural planning, and many others. While some of the materials use

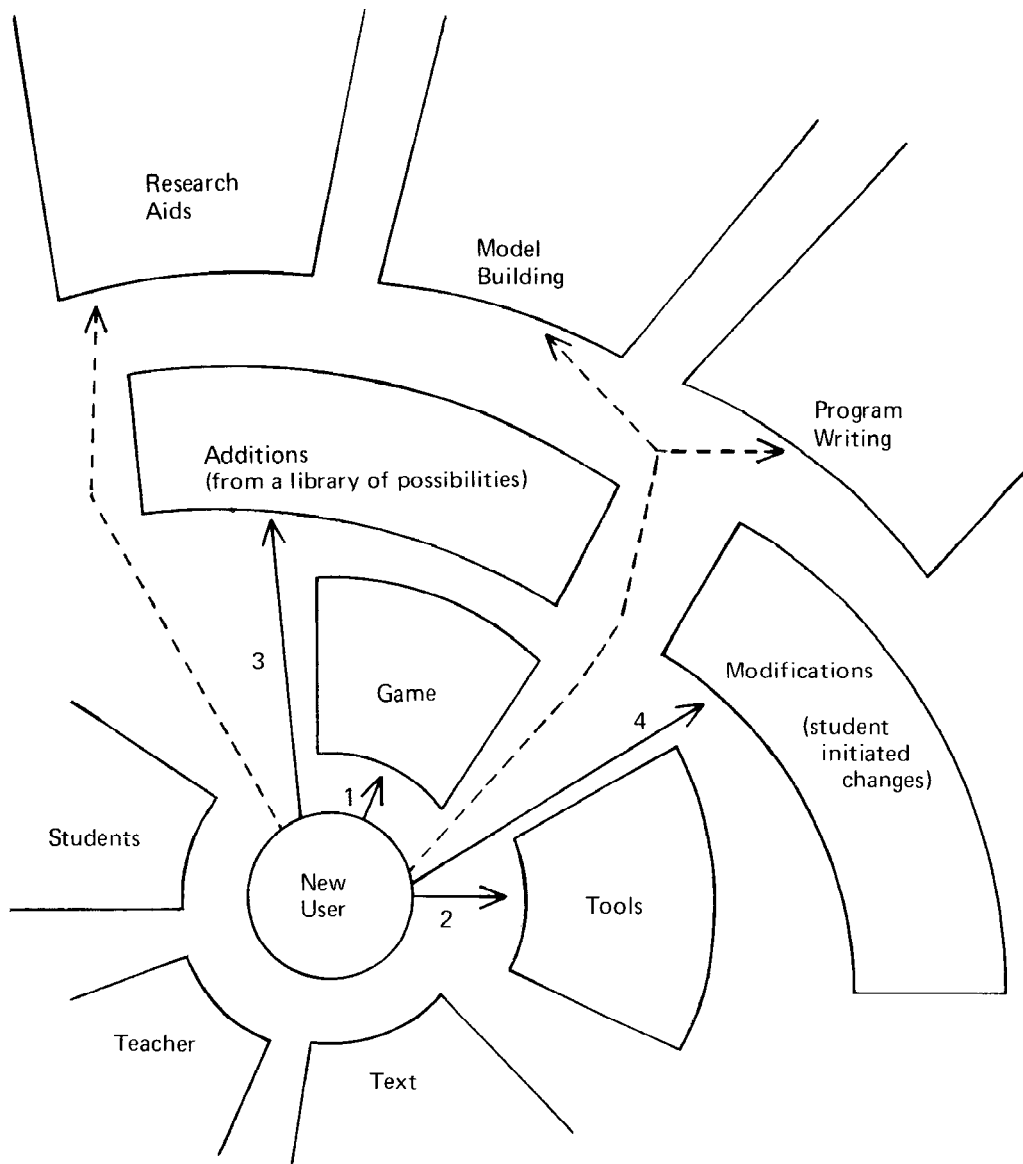


Fig. 4. An open-ended approach to student involvement with computing activities.

the computer as an information processing device, others use it as a presentation medium in competition with less expensive modes such as books, films, or video tapes.

Strategies of instruction associated with computer use (the name "teachware" has been proposed) are at an early stage of development. Guidelines for writing instruction-related computer programs have been derived from psychological and educational research, but most developers work from a "common sense" analysis and by trial and error. Some basis for

a new science of instruction can be found in the research programs of Robert Glaser at the University of Pittsburgh, Robert Gagne at Florida State University, and M. David Merrill at Brigham Young University.

The costs of using various operational or experimental computer instruction systems and languages vary considerably. Figures reported by manufacturers and research projects range from \$0.15 to \$30.00 per student per hour. Some of the differences can be attributed to variations in assumptions about how

COMPUTER-ASSISTED LEARNING AND TEACHING

many effective student hours can be scheduled in a month or a year; whether the equipment is rented, leased, or purchased; and how much time will be spent in utility jobs, preventive maintenance, or repair; and what accounting methods should be used. About \$3.00 per hour was a typical charge for interactive use of computing within educational institutions in 1975. Two major demonstration projects, the PLATO Project directed by Donald Bitzer at the University of Illinois (Urbana) and the TICCIT Project managed by Kenneth Stetten at MITRE Corporation (McLean, Virginia), plan to achieve a cost per student hour of less than \$0.50, covering all costs including student's use of a user terminal, communications to the computer, and curriculum materials.

COMPUTER CONTRIBUTIONS. The value of computer assistance for self-instruction depends on many factors: organization of the subject matter, the purposes of the author or institution, convenient means for interacting with the subject, and the characteristics of the student. Self-study material in text format has been adapted for computer presentation, with the following computer contributions proposed: First, the machine evaluates a response constructed by the student (the author must provide a key or standard); an automated procedure prints out discrepancies, tallies scores, and selects remedial or enrichment material. Second, the machine conceals and, to some extent, controls the teaching material so that the author can specify greater complexity in a strategy of instruction and assume

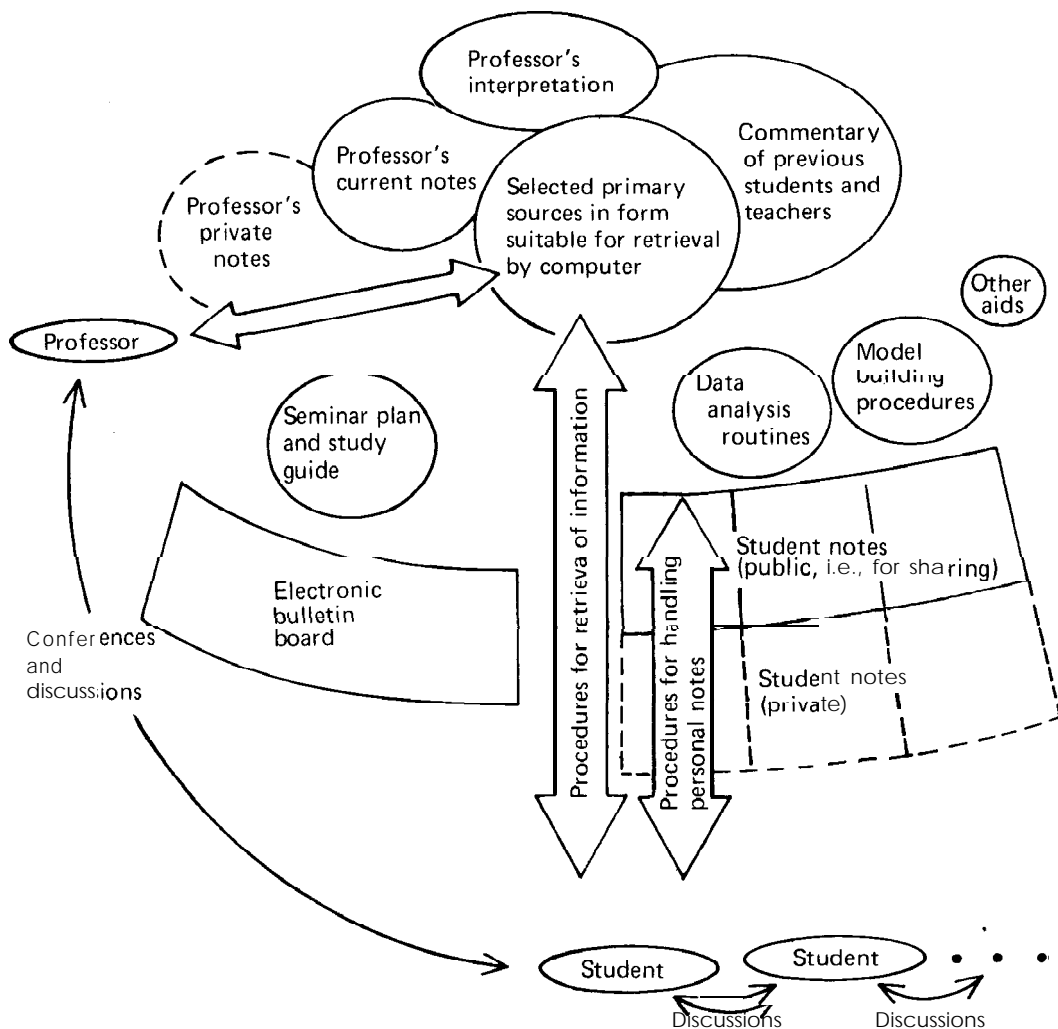


Fig. 5. A dynamic information system for scholar and learner.

more accuracy in its execution than is possible when the student is expected to find his way through the branching instructions in the pages of a large booklet (the scrambled text format for programmed instruction). Third, the computer carries out operations specified by the student, who uses a simple programming language or computer-aided design system. Fourth, the author or researcher obtains detailed data on student performance (and perhaps attitude) along with a convenient summarization of student accomplishment ready for interpretation. Fifth, the author is able to modify his text on the basis of student use and prepare alternative versions with relative ease.

Prepackaged self-instruction can be replaced by a dynamic information system that serves as a common working ground for a scholar and a learner; they share a computer-based, primary-source "textbook," continually updated by the scholar and occasionally annotated by each student who uses it (Fig. 5). Prototype systems have been demonstrated: HYPERTEXT was conceived by Theodor Nelson (then at Vassar College) and implemented by Andries van Dam and others at Brown University; and SPIRES was designed by Edwin Parker and others at Stanford. In a similar way, an automated information system helps a learner and teacher share a common working environment for hypothesis testing. The environment is sometimes artificial, as in computer simulation of physical and social processes (e.g., a model of evolution), and sometimes real in the sense of actual data from experiments (e.g., election returns or radiation measures). Increased access to information processing tools is perhaps the most important contribution of computers to instruction and learning. Many such activities, perhaps not called computer-assisted instruction, demonstrate viable alternatives to strictly specified instructional strategies for computer use. For example, students in sociology retrieve and summarize information obtained from large-scale surveys and test hypotheses that might never have been conceived by those who executed and reported the survey. Students in physics test lens designs according to a detailed model of aberrations and corrections, perhaps finding variations on standard lens designs which better serve a particular photographic or instrumentation purpose.

Whatever the technique or philosophy of computer use, the extent of use will ultimately be determined by judgments of appropriateness by subject experts, effectiveness observed from records of student performance, and costs that must be met by administrators of schools or training programs.

Costs may be reduced and quality of instruction improved through computer aids for preparation and revision of material, self-modifying strategies of instruction, and automatic assembly of additional teaching samples or testing situations.

Some of the limitations imposed by present computer technology involve high cost and unreliability of processing lengthy verbal constructions, and inability to interpret bodily gestures or vocal intonations. Computing costs are decreasing even while capabilities are increasing, but one of the most difficult problems remaining is lack of organization of the subject matter. Somehow, human teachers manage to be reasonably successful in spite of vague goals and material poorly organized for learning; instructional computing (and educational technology in general) seems to require specific text materials and clear guidelines (prepared by curriculum experts) for successful use.

Major Approaches

EDUCATIONAL TECHNOLOGY. Educational technology and instructional psychology have been the main sources of one kind of development activity. IBM's Coursewriter programming language, one of the earliest languages for authors of computer-based lessons, characterizes this first approach to computer use. The software has built into it an implicit logic of instruction, requiring the author to fit text and key words into the following pattern: (1) the computer program presents information to the student; (2) the computer program then asks a question and waits for a response from the student; (3) the program scans a short textual response and classifies the response as right or wrong according to key words identified within it; and (4) if the student's response matches an anticipated wrong answer, the program displays a corrective hint, and if nothing was recognized, it offers a general hint. Many instances of this approach can be characterized as the computerization of programmed instruction. Careful development of a total curriculum for elementary school mathematics and reading was carried out by teams of authors at Stanford University directed by Patrick Suppes and Richard Atkinson. The same approach has been used at the University of Texas for college mathematics, at Pennsylvania State University for education methods courses, and at the U.S. Naval Academy for college physics.

In some curriculum development projects the content has been assembled in files separate from the logic of the computer program (the strategy of instruction). Elements of the curriculum can thereby

COMPUTER-ASSISTED LEARNING AND TEACHING

be varied without rewriting many lines of instructions to the computer, and different strategies can be tried on the same file of learning materials. This arrangement helps the instructional psychologist give full attention to the design of effective instructional strategies and helps the subject expert avoid the distraction of programming procedures. In fact, this approach is generally pursued by a team, with each member contributing different expertise. Authoring teams organized by C. Victor Bunderson at Brigham Young University are working on materials for community college courses in mathematics and English for the TICCIT system under development by Mitre Corporation.

Problems faced by the educational technology approach to computer use result from the high cost of the computer as a primary medium for exposition of learning materials, the difficulties of accurately identifying unconstrained input (text, algebraic expressions, drawings, spoken expression, etc.), and the lack of a well-developed theory of instruction. These and other obstructions to more widespread use of computers for instruction have been discussed in a report by Anastasio and Morgan (1972).

DISCIPLINES AND CURRICULUM. Discipline-oriented use of the computer has until recently been pursued by many institutions quite separately from the educational technology developments. Dartmouth College provides a prime instance of spreading computer uses throughout a college curriculum. The University of California at Irvine uses computing extensively in physics courses. Six annual conferences have been held on the topic of computers in the undergraduate curriculum; the sites have been the University of Iowa (1970), Dartmouth College (1971), Georgia Institute of Technology (1972), the Claremont Colleges (1973), Washington State University (1974), and Texas Christian University (1975). Regional computing services, conferences, and newsletters have been established to serve the needs of colleges throughout a region. In contrast to the educational technology approach, the teacher as subject expert in the discipline approach assumes the central role in determining computer use, creating materials, and persuading colleagues to use them. Computing activity is likely to include more student initiative in solving problems and more problem-orienting program packages than does expository material. Student use of simulation and modeling tools is favored; one goal is to adapt the scholar's research tools to student use.

The discipline approach to computer use has many problems; among them are: sparse user documentation for instruction-related computing activ-

ities that are worthy of widespread use; lack of economic and professional incentives for the production and dissemination of programs and related materials; and difficult procedures for review and validation of programs. The National Science Foundation has sponsored a consortium of regional computing services (called CONDUIT) to explore solutions to these problems. Publications and program exchange activities with a discipline orientation are being pursued within professional societies.

COMPUTING AND INFORMATION SCIENCES. Some researchers suggest that major advances in instructional use of computers will occur through significant developments in artificial intelligence, natural language processing, speech recognition, and extensible programming languages. Although information scientists typically are more interested in their own disciplines and related research topics than in educational techniques and practice, the tools developed may be useful to others. The results of computer science research may be an important source of suitable models for instruction strategies, information structures, and representations of knowledge. The work of a dozen computer science projects is referenced in a summary of directions for research and development on computers in the instructional process (Zinn, 1972). Projects giving particular attention to educational applications are located at the Massachusetts Institute of Technology, Bolt, Beranek, and Newman, Carnegie-Mellon University, the University of Texas, California Institute of Technology, and Stanford University. In addition to the tools to be borrowed from computing and information sciences, new models of human learning and information processing may be obtained.

The information science approach has not yet produced many operational systems. Development of techniques and materials is very costly and time consuming; the resulting applications are expensive in execution with students; skill in use of the specialized techniques is not easily acquired by persons outside computer science. Nevertheless, the projects based in computing and information sciences continue to provide important indicators of future resources which may be essential to success of computers in education. Furthermore, a project at the University of Illinois directed by Jurg Nievergelt has adapted ideas from computing and information sciences for economical execution so that the computer-based materials can be used by over 2,000 students per semester in a set of introductory courses. Because the courses are in computer science, the authors combine a detailed knowledge of the specialized techniques with considerable expe-

rience in teaching the subject.

COMPUTING TECHNOLOGY, ENGINEERING, AND "COMMON SENSE". A fourth category includes all other approaches, particularly those characterized by the engineering of a helpful technology, perhaps involving some combination of the first three approaches. Engineers at the Computer-Based Educational Research Laboratory, University of Illinois, designed and built a computer-based education system (PLATO, Fig. 6) which is intended to be convenient for: educational technologists presenting programmed instruction, instructional psychologists conducting research on teaching and learning, professors preparing a computer presentation of a lecture or laboratory, and computer specialists building information processing aids for learning and scholarly work. Specialists in computers and education at Bolt, Beranek, and Newman (led by Wallace Feurzeig) and the Massachusetts Institute of Technology (notably, Seymour Papert) have devised various programming languages (Mentor, Telcomp, Stringcomp, and Logo) and equipment (computer-controlled "turtle," music player, etc.) for computer-related learning activities.



Fig. 6. Student terminal used with the PLATO computer-based education system, which includes a flat panel on which characters and line drawings are combined with rear projection of color images. (Courtesy of Computer-Based Education Research Laboratory, University of Illinois.)

Two other groups evidence a similar philosophy but offer different approaches to creative student work: the Soloworks Project directed by Thomas

Dwyer at the University of Pittsburg and the Learning Research Group directed by Allen Kay at the Palo Alto Research Center of Xerox Corporation. In each case, children write simple programs for controlling robots, drawing and animating pictures, generating speech and music, and the like. Their interest in enhancing such capabilities motivates a new approach to mathematics and heuristics in which programming languages provide a powerful conceptual framework. The M.I.T. group has given particular attention to teaching children how to think, reason, and solve problems. These projects and others having an engineering approach are described in the proceedings of the World Conference on Computer Education (Scheepmaker and Zinn, 1971).

Trends. A major trend in the design of computer-based exercises is a shift from programmer to learner control. The designer of the exercise invests less effort in a careful diagnosis and prescription accomplished by some automated instructional strategy, and instead provides information from which the student can derive his own diagnosis from alternative interpretations and guidance. The student can assemble prescriptions specific to his own situation.

Increased use of graphics is seen in many projects. Pictures are an important component of the learning process, and computer-drawn pictures can add to the responsive uses of computing. For many topics the picture is a valuable way of representing complex relationships derived by the computer.

Computer-based education systems and designers of materials are providing an increasing variety of functions for the user. More attention is being given to interaction between student and computer program, not simply to provide a quick reply to some question, but to increase the actual responsiveness of the system to the particular input. The SCHOLAR system designed by the late Jaime Carbonell at Bolt, Beranek, and Newman is a prime instance. Machine responses are increasingly dependent upon the commands and questions and answers constructed by the student, and the lessons are designed in a way that helps the student respond to information provided by the computer.

A very important trend concerns the role of the machine from the perspective of the individual using it. The teacher is now more likely to see computer-managed instruction as an aid to human management than as a replacement for it. Learners view the machine more as an aid to learning than as a drill master.

COMPUTER CIRCUITRY

Naturalness of communication between learner and system is being improved day by day. Computer-based learning exercises are achieving increased relevance for the subject being studied, and the nomenclature and conventions that have to be learned in order to use the system tend to be essential to the study of the topics rather than peculiar to the requirement of the computer as a medium of presentation.

Although these advances may be quite impressive to observers of educational research and development, dissemination of sound practices throughout educational systems will be a long time coming. The influence of the new technology within education will, as usual, lag behind the impact of the new technology on society.

REFERENCES

1968. Silberman, Harry F., and Robert T. Filep. "Information Systems Applications in Education," in Carlos A. Cuadra (Ed.), *Annual Review Of Information Science and Technology*, Vol. 3. Chicago: Encyclopaedia Britannica, Inc., pp. 357-395.
1971. Scheepmaker, Bob, and Karl L. Zinn (Eds.). *IFIP World Conference on Computer Education 1970*, Amsterdam: International Federation of Information Processing.
1972. Anastasio, Ernest J., and Judith S. Morgan. *Factors Inhibiting the Use Of Computers in Instruction*. Princeton, N.J.: EDUCOM (Interuniversity Communications Council, Inc.)
1972. Levien, Roger E. *The Emerging Technology: Instructional Uses of the Computer in Higher Education*. New York: McGraw-Hill.
1972. Zinn, Karl L. "Computers in the Instructional Process: Directions for Research and Development," *Communications Of the ACM*, Vol. 15, No. 7 (July), pp. 648-65 1.
1973. Vinsonhaler, John, and Robert Moon. "Information Systems Applications in Education," *Annual Review Of Information Science and Technology*, Vol. 8. Chicago: Encyclopaedia Britannica, Inc.
1974. Zinn, Karl L., Mario Refice, and Aldo Romano (Eds.). *Computers in the Instructional Process: Report Of an International School*. Ann Arbor: EXTEND Publications.

K. L. ZINN

COMPUTER CIRCUITRY

For articles on related subjects see **GENERATIONS, COMPUTER;** and **INTEGRATED CIRCUITRY.**

For articles on related terms see **ADDERS;** **BINARY CODED DECIMAL, NATURAL;** **ENIAC; MARK I; SEQUENTIAL MACHINES;** and **SHIFTING.**

Although the development of digital computers can be traced back to Charles Babbage, who conceived a mechanical machine with toothed wheels to perform arithmetic processes, electrical principles were first utilized in a computer through electro-mechanical relays by the Bell Telephone Laboratories in 1940. The first general-purpose computer with relays was the Harvard Mark I which was jointly developed by International Business Machines Corporation (IBM) and Harvard University in 1944 (Huskey and Korn, 1962). The first electronic computer implemented by vacuum tubes was ENIAC, built in 1945 at the University of Pennsylvania for the US. Army to solve ballistic problems. In this article, we will survey the circuits (and their general characteristics) used in various generations of computers, present some commonly used computer circuits, and discuss the trend of future computer circuitry.

Circuitry in Various Generations of Computers.

Modern digital computers contain thousands of logic circuits for their arithmetical and logical operations and for their control of data flow, all composed of only a few different kinds of elementary circuits. Through boolean algebra manipulations, any complex logic function can be reduced to three primary operations: **AND**, **OR**, and **INVERT** operations. In the **AND** operation—also called "logical multiplication" (or "product") or "intersection"—the output will be up or in a high-voltage position if and only if all the inputs are up. In the **OR** operation—also called "logical sum" (or "sum") or "union"—the output will be up if one or more of its inputs are up. In the **INVERT** operation—also called "negation," "complementation," or "NOT" operation—a logic circuit has only a single input. Its output will be up only when the input is down, or vice versa. A set of logic circuits capable of performing these three logical operations is said to be *functionally complete*. There are two other well-known functionally complete sets (Kohavi, 1970;

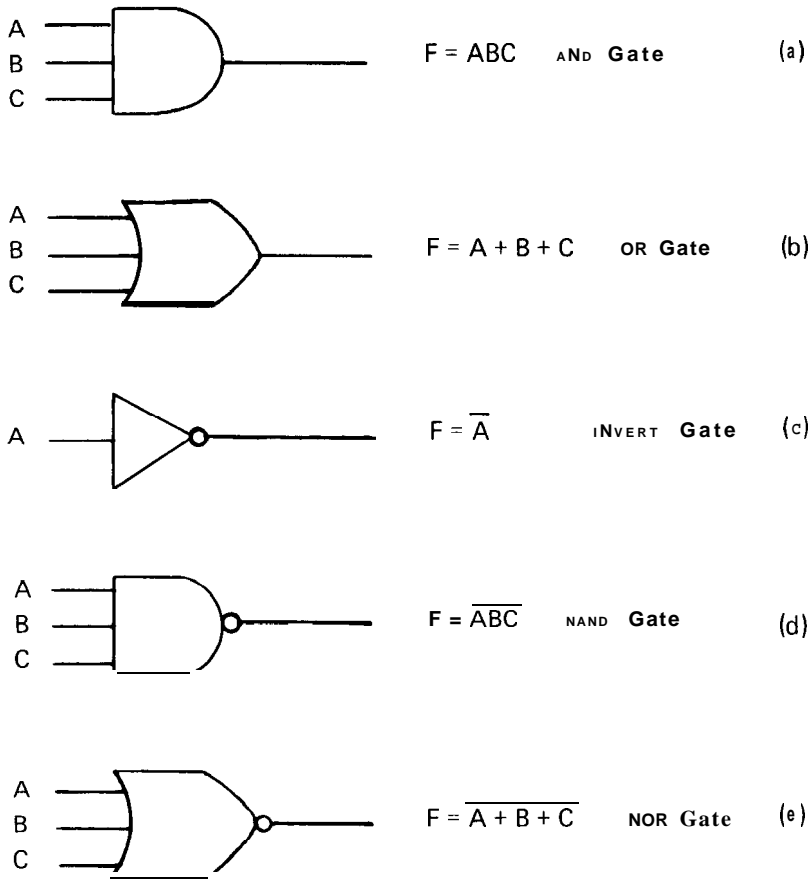


Fig. 1. Logic symbols.

Torng, 1972). One consists of only the **NAND** circuit which is actually an **AND** circuit with its output inverted. The other consists of only the **NOR** circuit which is an **OR** circuit with its output inverted. With either circuit, all logical functions of a modern digital computer system can be implemented. 'The **AND**, **OR**, **INVERT**, **NAND**, and **NOR** circuits are shown symbolically in Fig. 1. A circuit performing any of these basic operations is usually referred to as a "gate."

Over the years, logic circuit configurations and the technologies to produce them—as well as the architecture, complexity, software sophistication, and computing performance of computers—have gone through an enormous evolutionary process. So striking has this evolution been that computers are universally classified according to the concept of generation. Although there is not always general agreement among those in the computer community,

a computer generation has been widely defined in terms of the logic technology and the structure of the active logic devices. The *first generation* computers used vacuum tubes, mostly triodes and pentodes, and spanned the years from approximately 1945 to 1958. The *second* generation computers used transistors, starting about 1958 and ending about 1965. Since about 1965, computers have contained integrated circuit versions of transistor circuits and have commonly been called *third generation* computers. The *fourth generation* computers have yet to be clearly defined, but they will probably use medium or large-scale integration circuits extensively. Thus, several logic circuits will be described here according to the computer generations with which they are most closely associated. Actual implementation of interesting logic circuits will also be presented.

FIRST-GENERATION COMPUTERS. The first-generation electronic computers were primarily char-

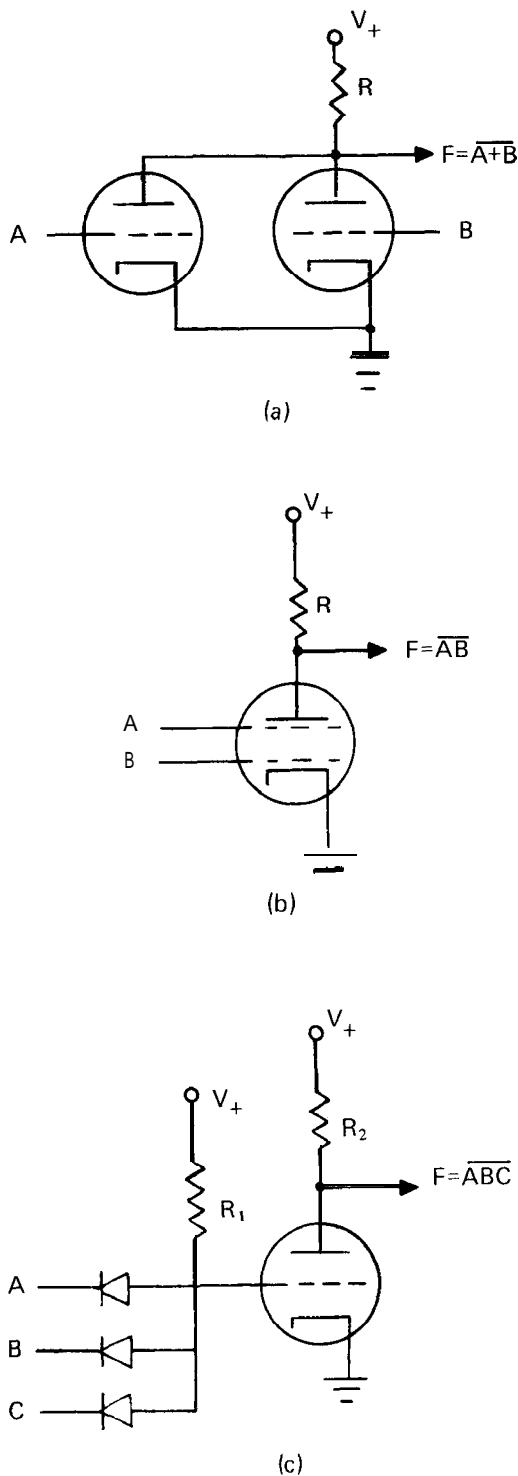


Fig. 2. First-generation circuits. (a) NOR gate; (b) NAND gate; (c) INVERTER.

acterized by a logic technology utilizing vacuum tubes. A **NOR** gate composed of two triodes is shown in Fig. 2(a). If one or two grid inputs (A, B) are at logical 1 (i.e., high-voltage state), one or both triodes will conduct **current**, causing a potential drop across the load resistor R and an output of logical 0 (i.e., low-voltage state); otherwise, the triode will not conduct and the output will be a logical 1. With only one triode, the circuit performs an **INVERT** function. The twin triodes performing **NOR** functions were later put into one single vacuum chamber.

Another important vacuum tube for computer applications was the **pentode**. The high-impedance control and suppressor grids of a **pentode** were used as input grids, while the low-impedance screen grid was usually not used. A **pentode** circuit without the screen grid is shown in Fig. 2(b) and behaves as a **NAND** gate. If either grid input is a logical 0, the **pentode** will not conduct and the output will be a logical 1. If both grid inputs are a logical 1, the **pentode** will conduct and the output will be a logical 0. A **pentode INVERT** circuit is formed by setting one of the inputs constantly at a logical 1.

Since either the **NAND** gate or the **NOR** gate is sufficient to generate any boolean function, either the twin-triode or **pentode** circuit is sufficient for realizing more complex circuits. However, the voltage levels for either circuit do not have compatible input and output requirements. Therefore, **voltage-level** restoring circuits have to be introduced for cascading either the twin triode or **pentode** logic circuits. A resistance-voltage divider circuit can be used for voltage-level restoring, but resistor tolerances may introduce imprecise logical levels.

Besides the high cost, computer circuits using vacuum tubes had other shortcomings. Vacuum tubes were limited by their large physical size, which introduced substantial transmission delays. Power consumption was high, so that cooling requirements were also high. Furthermore, they had a rather limited lifetime and a gradual deterioration property, which restricted the practical size of a system that could be seriously contemplated. Consequently, the complexity of the first-generation computers was quite limited. ENIAC contained approximately 18,000 vacuum tubes and was the largest vacuum tube computer ever attempted. With so many vacuum tubes, reliability was quite poor.

A major advance in this technology was the practical application of germanium diodes as logic gates to reduce the required number of vacuum tubes. For example, a multi-input **NAND** gate could be formed from a multi-input diode **AND** gate followed by a triode or a **pentode INVERT** circuit as

shown in Fig. 2(c), and thus replace an equivalent circuit requiring several vacuum tubes. The Whirlwind I computer (1951) built by Massachusetts Institute of Technology had a speed of 20,000 operations per second, the fastest computer of its time. This computer required only 5,000 vacuum tubes, mostly pentodes, but there were 11,000 diodes. The IBM 701 computer (1953) had 4,000 vacuum tubes, mostly twin triodes, and 13,000 germanium diodes.

SECOND-GENERATION COMPUTERS. The logic circuits used in second-generation computers were primarily discrete circuits using transistors. Although invented in 1948, the transistor required a decade of development effort before it became a superior alternative to vacuum tubes. Transistors are faster, smaller, more reliable, and dissipate less power than vacuum tubes. A faster operating device results in greater computing power. A smaller device with less power dissipation permits greater packaging density with shorter interconnections, which reduce stray reactance and a shortening of transmission delays. A more reliable device permits a larger and more complex computer to be successfully built.

There are many ways and configurations to implement logic circuits with semiconductor diodes, resistors, and transistors. The transistors in this second generation were primarily bipolar transistors, which means that carriers of both polarities, electrons and holes, are involved to form the total current. No single transistor configuration was superior to all others in all respects. Some of the more important attributes of a logic circuit include speed, fan-in/fan-out capability, noise immunity, noise-generation/stabilization properties, operating temperature range, power dissipation, and cost. The more widely used types of circuits include RTL (resistor-transistor logic), DTL (diode-transistor logic), TTL (transistor-transistor logic), and ECTL (emitter-coupled transistor logic), which are shown in Fig. 3.

RTL (Resistor-Transistor Logic). The basic RTL circuit is shown in Fig. 3(a). The RTL circuit is a simple and inexpensive logic circuit. Resistors R_1 , R_2 , and R_3 form an OR gate. The transistor T along with its load resistor R_4 forms the amplifier-inverter section of the circuit in a manner similar to the triode shown in Fig. 2(c). The RTL circuit is therefore a NOR gate and is relatively slow.

DTL (Diode-Transistor Logic). The basic DTL circuit is shown in Fig. 3(b). Speed, fan-out capability, noise immunity, and power dissipation are good. When one or more of the inputs (A, B, C) are at logical 0, or low-voltage state, current will flow

from V_+ through resistor R_1 into the inputs. Point P as well as Q will be in a low-voltage position. The transistor will be off and the output F will assume a high-voltage, or logical 1, state. Only when all inputs are in logical 1, or high-voltage, state will the current then be directed to flow through R_1 , two diodes in series, and R_3 into V_- . Point Q will now be at a higher voltage level to turn on the transistor. Current will now be allowed to flow through the transistor, and the output F will assume a low-voltage, or logical 0, state. The two diodes in series are used in order to get the correct voltage level at point Q . The output can go to logical 0 state only when all inputs are logical 1. The DTL circuit, therefore, performs a **NAND** function.

TTL (Transistor-Transistor Logic). The basic TTL circuit is shown in Fig. 3(c). The circuit is also a **NAND** gate and is capable of significantly higher speed operation than the RTL and DTL circuits. When either input A or input B , or both, are at logical 0 state, there will be sufficient base-to-emitter voltage difference so that either T_1 or T_2 , or both, will be turned on. Point P , which connects to the collectors of both transistors, will assume a low-voltage state to turn off T_3 . There will be "on" current flowing through R_2 and the output will be high, or at logical 1. When and only when both inputs A and B are high, both transistors T_1 and T_2 will be "off" and point P can return to a higher voltage level to turn on T_3 . Output F will come to a low-voltage state when T_3 is "on" and current flows through R_2 .

ECTL (Emitter-Coupled Transistor Logic). The basic ECTL circuit is shown in Fig. 3(d) and is potentially the fastest transistor logic circuit available. All the transistors in ECTL operate in a nonsaturating mode in order to attain high speeds. The emitter current passing through R_3 is essentially constant with the current passing through R_2 when any of the inputs is a logical 1 or passing through R_2 when every input is a logical 0. Transistor T_4 establishes the reference voltage for the logical 0 and logical 1 states of the input transistors T_1 , T_2 , and T_3 . The input transistors in combination with the reference transistor act as a differential amplifier having good common mode rejection of power-supply line noises. Also, both a **NOR** output \bar{F} and an **OR** output F are available, yielding complementary gating functions respectively.

Typical second-generation computers using these circuits include the IBM 7000 series (first delivery 1960) and Burrough B200 series (first delivery 1961).

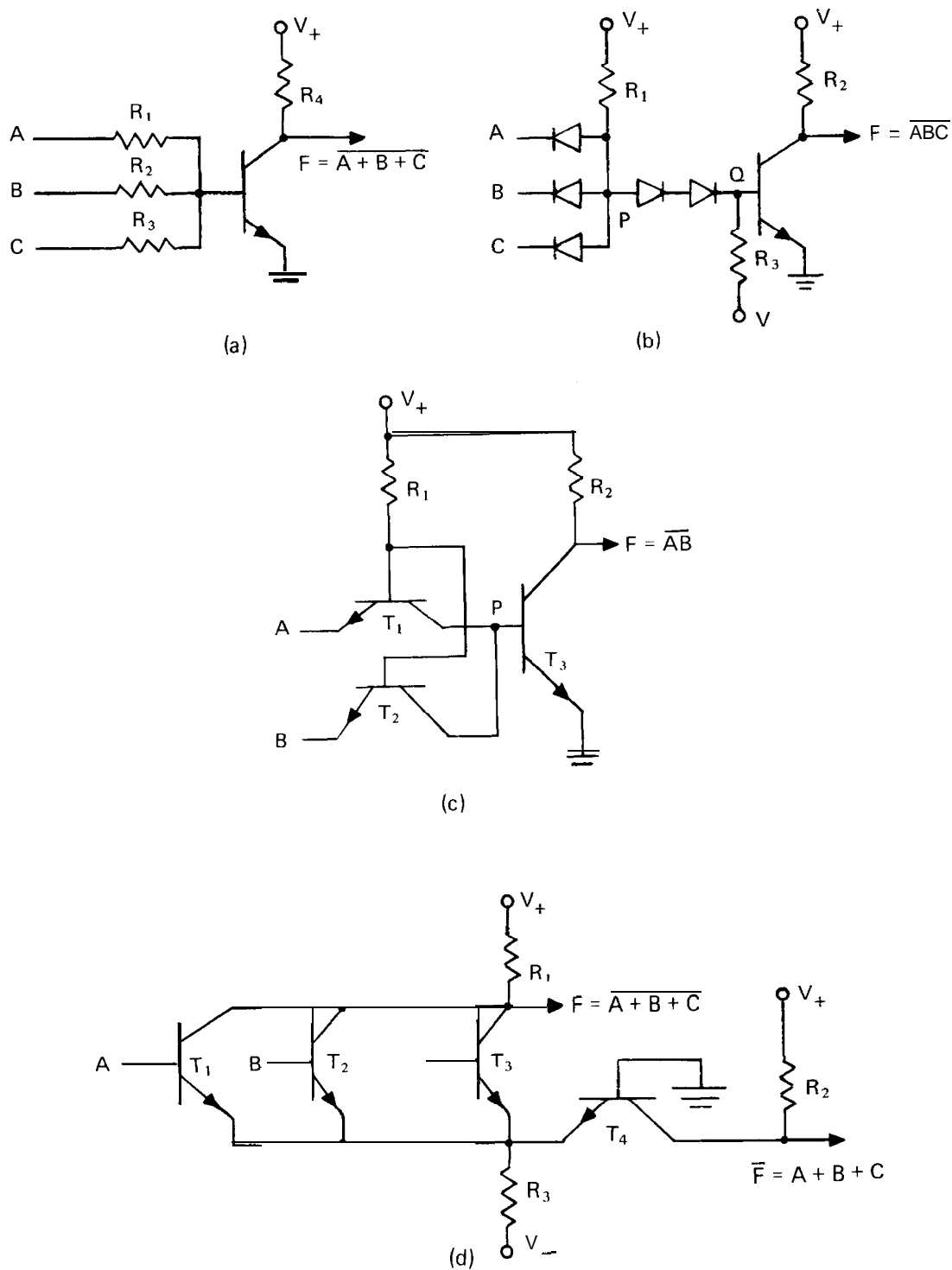


Fig. 3. Second-generation logic circuits. (a) RTL circuit; (b) DTL circuit; (c) TTL circuit; (d) ECTL circuit.

THIRD-GENERATION COMPUTERS. A computer using vacuum tubes was considered a first-generation computer, and one using transistors was considered a second-generation computer. However, the distinction between second- and third-generation computers is not so clear-cut. Those manufacturers using integrated circuits tend to believe that the use of integrated circuit (IC) technology should be the criterion for distinguishing third- from second-generation computers. On the other hand, those manufacturers still using discrete component technology tend to believe that performance would be a better measure. Thus, Control Data Corporation 6000 series computers, which were implemented by discrete component circuits, should certainly be classified as third-generation computers, according to their performance. Furthermore, the IBM 360 family of computers is considered third-generation, although it utilizes hybrid circuit technology that is partially integrated circuit technology and partially discrete device technology. Nevertheless, from the computer circuitry point of view, integrated circuit technology will be used to define a third-generation computer whether the circuitry is fully or only partially integrated.

The logic circuits used in a third-generation computer have the same basic circuit configuration as those in second-generation computers. In integrated circuits, the transistors, diodes, and resistors

are all fabricated simultaneously on a silicon wafer. Cost differentiation among these devices, which is of utmost importance in discrete components, is not significant in integrated circuits. TTL circuits and ECTL circuits, which use more transistors, can be fabricated at about the same cost as RTL or DTL circuits. Consequently, the integrated circuit versions of logic circuits tend to have more transistors, are more complex, and have better performance at lower cost.

Hybrid DTL. The IBM 360 family of computers (first delivery 1965) used hybrid circuits. One aspect of this design is that the transistors and diodes were encapsulated in a protective layer of glass so that a hermetic seal was unnecessary. Resistors are fabricated as thin-film devices and metallization patterns make the substrate interconnections. The logic substrate contains a DTL circuit, which is given in Fig. 4. Basically, the circuit is still a **NAND** gate when **A**, **B**, and **C** are used. The **X** input lead is called an "expander" and serves to connect additional diodes that may be added to increase fan-in. The **R** input lead permits an **OR** coupling circuit to be added to the logical operations of the circuit. Some of the advantages of this technology are better quality and reliability for both active and passive components, high component density, and high-speed performance.

Integrated TTL. The integrated version of TTL

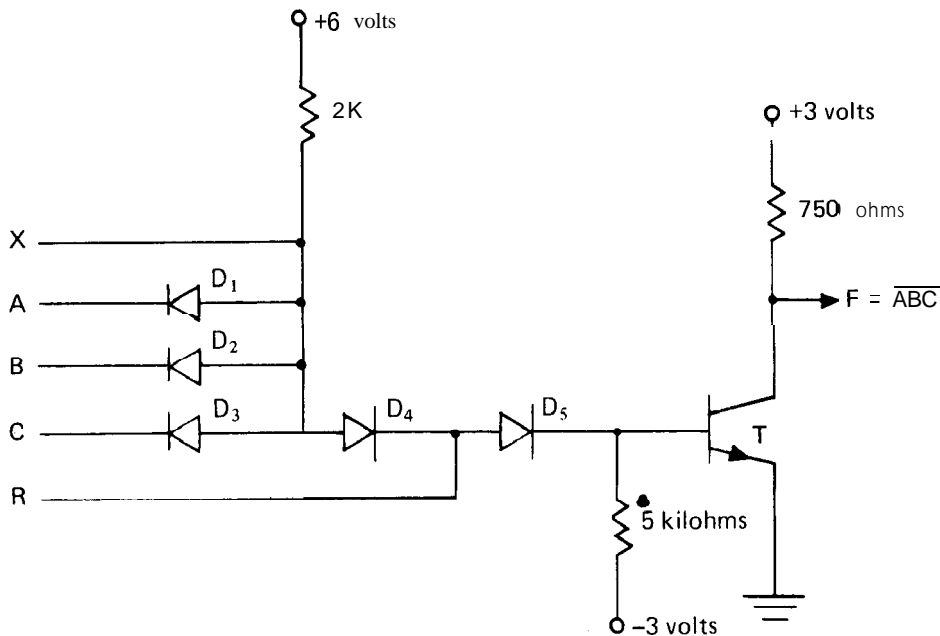


Fig. 4. The DTL circuit of the IBM 360 (SLT).

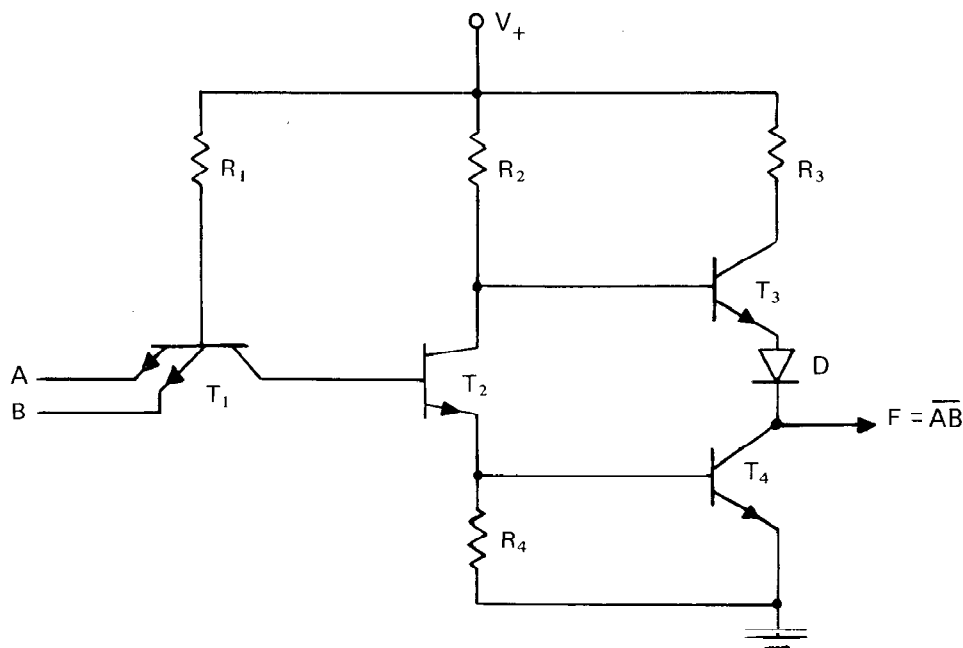


Fig. 5. An integrated TTL circuit.

shown in Fig. 5 is quite different from the discrete component TTL shown in Fig. 3(c). The input transistors T_1 and T_2 in Fig. 3(c) are combined into a single transistor with double emitters. The output circuit typically uses active devices instead of passive resistors to pull up (T_3) and pull down (T_4) the output so that better and faster switching rates can be maintained even when the circuit is driving long lines of capacitive loading.

The IC versions of DTL and ECTL have no significant differences from their counterparts in discrete components, and require no further discussion.

Integrated Field-Effect Transistor Circuits. After the mid-1960s, the MOSFET (metal-oxide-semiconductor field-effect transistor) has competed seriously with the bipolar transistor for logic applications. MOSFET is a unipolar device whose current is transported by carriers of one polarity only. MOSFET logic circuits are not competitive in high-speed logic applications (Crawford, 1967), but they are challenging bipolar circuits because of their lower cost, lower power consumption, and high density. Typical MOSFET logic circuits are shown in Fig. 6. A MOSFET device can be either an N-channel or a P-channel device. In the former, the current is carried by electrons; in the latter, by holes.

General Characteristics of Various Types of Computer Circuitry. All electronic logic circuits have certain characteristics (Lo, 1967) in common:

1. **Directivity.** The ability of the logic circuit to insure that the input signal is not affected by the output signal. A unilateral data flow is thus obtained.
2. **Isolation.** The ability of a logic circuit having more than one input to supply isolation among the inputs.
3. **Fan-in, fan-out.** The ability of the logic circuit to provide multiple inputs to enable signal interaction, and the ability to provide power gain so that the output can drive more than one similar logic circuit as its load. Power gain guarantees that an input signal will not deteriorate after propagating down a data path.
4. **Quantization.** The ability of the logic circuit to preserve the binary "0" and "1" signals distinct from input to output is called "quantization." Since electronic circuits can be made economically with two easily distinguishable states (high-impedance and low-impedance), binary quantization is most suitable for digital computer application. Decimal and other number systems are generally represented

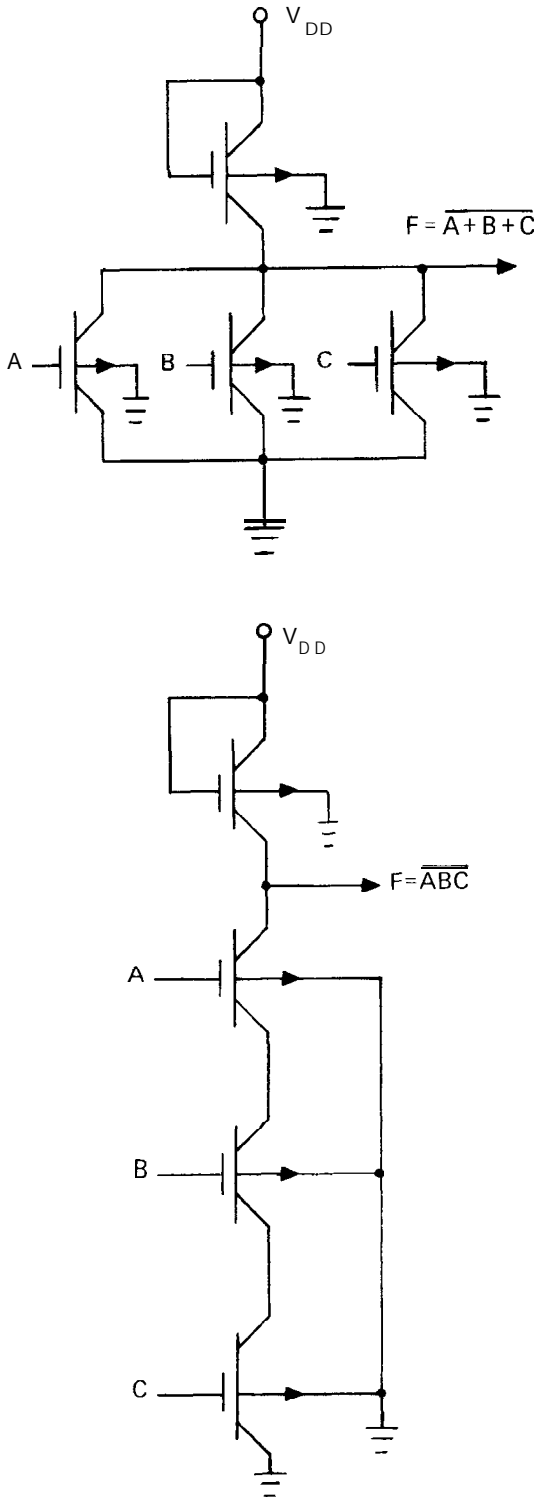


Fig. 6. MOSFET NOR and NAND circuits.

by a binary-coded form, such as BCD (binary-coded decimal).

The four general characteristics of logic circuits listed above can best be explained further with a typical logic circuit such as the DTL shown in Fig. 3(b). The property of directivity specifies that the output F is controlled by the inputs A, B, C , but not vice versa. This property is achieved through the characteristic of the transistor. Its base controls the "on" or "off" of the transistor as well as the voltage level at the output F . The output, however, does not control the base voltage.

The property of isolation specifies that a voltage swing of any one input, such as A , has no effect on the other inputs, such as B and C . Between input A and input B are two back-to-back diodes. No matter which way A swings with respect to B , one of these two diodes will be reverse-biased. Therefore, input B is isolated from, and will not respond to, the swing of input A . Obviously, the isolation between A and C works the same way.

The fan-in of the DTL circuit shown in Fig. 3(b) is three, i.e., the number of inputs driving the output. The ability to support fan-in is essential for performing the **AND**, **OR**, **NAND**, or **NOR** functions. The fan-out of a DTL circuit refers to how many similar DTL circuits can be driven from the output point F of this circuit. This fan-out number is limited by the total current that is allowed to flow through the transistor without appreciably raising the voltage level of F . The total current includes that from V_+ flowing through R_4 and those from the driven DTLs.

The property of quantization in DTL circuits refers to the fact that the voltage swing at the output F with any allowable fan-out should be the same as the voltage swing of the inputs. The quantization is obtained by properly designing the three resistors with the two voltage supplies of the DTL circuits.

Some Commonly Used Computer Circuitry. Computer circuitry can be divided into combinational circuits and sequential circuits. We will discuss them separately.

COMBINATIONAL CIRCUITS. A combinational circuit is a logic circuit whose output is determined solely by the states of its present inputs and is independent of the states of its previous inputs. Commonly used combinational circuits include majority logic, comparators, adders, and decoders. They are built from the basic logic gates shown in Fig. 1. Most of them are available in either bipolar or MOSFET integrated circuit modules from most semiconductor manufacturers. Some typical examples of combinational circuits quoted from semi-

conductor manufacturers' data sheets are described below.

Full Adder. A binary full adder will add two binary bit inputs (*A* and *B*) and a previous carry bit (*C_{in 1}*). The outputs will be a sum bit *S₁* and a carry bit *C_{out 1}*. The logic block diagram and the truth table are shown in Fig. 7.

BCD-to-Decimal Decoder. This circuit decodes a four-bit BCD (binary-coded-decimal) input to select one of ten outputs. The selected output is in the logical "0" state, and all the other outputs are in the logical "1" state. The logic layout and the truth table are shown in Fig. 8.

Many other combinational circuits available commercially are implemented by various technologies. The reader is referred to the data sheets or catalogs of the numerous semiconductor manufacturers.

SEQUENTIAL CIRCUITS. Logic circuits that can store digital information are classified as sequential circuits. In contrast to that of a combinational circuit, the output of a sequential circuit depends not only on the present input state, but also on the previous input states. From the circuitry point of view, a sequential circuit is different from a combinational circuit in that it has feedback paths that **connect** the outputs back to the inputs with proper phasing, time delay, **and** power gain. The feedback loops enable the sequential circuit to have

a stable state, which is self-sustaining even after the inputs are removed. Some most commonly used sequential circuits are described below.

Flip-Flop. A flip-flop is a basic storage element used for computer arithmetic operations. The logic layout, the MOSFET implementation, and the input-output truth table of a flip-flop are shown in Fig. 9. To understand the principle of operation of a flip-flop, let us assume that all MOSFETs are N-channel devices and that *T₁* and *T₂* are conducting. The output *Q* is at a low-voltage state (0), while \bar{Q} is at a high-voltage state (1) because the potential drop across *T₁* is much larger than that across *T₃*. The flip-flop is said to store a 0 under this condition. If, at this time, the reset gate *R* and the clock gate *C* are simultaneously energized by a positive pulse, *T₅* and *T₆* are made conducting, but the outputs *Q* and \bar{Q} will not change because *T₁* is still conducting. If, however, the set gate *S* and clock gate *C* are energized instead, *T₇* and *T₈* will start to conduct. Current can now flow from the source voltage *V_{DD}* through *T₃*, *T₇*, and *T₈* to ground. The potential drop in *T₃* will greatly increase and force the output \bar{Q} to take a low-voltage position. Through the cross-coupled feedback loop, the gate voltage of *T₂* is now too low to support the conduction of *T₂*. Since gate *R* also assumes a low-voltage position at this time, the previous current through *T₁* is now blocked, the potential drop across *T₁* is reduced

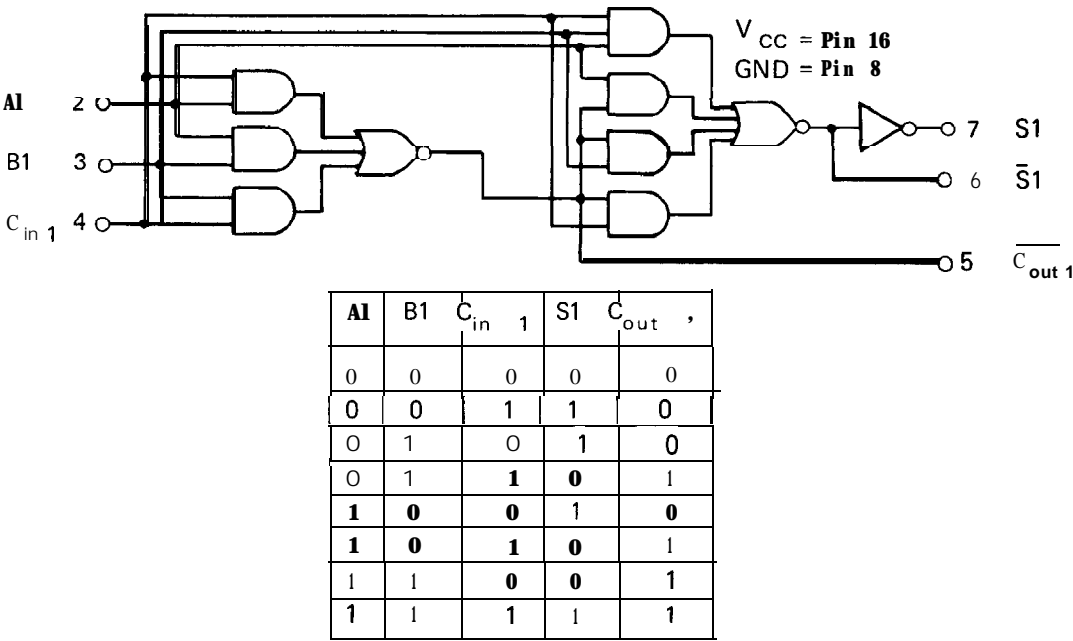
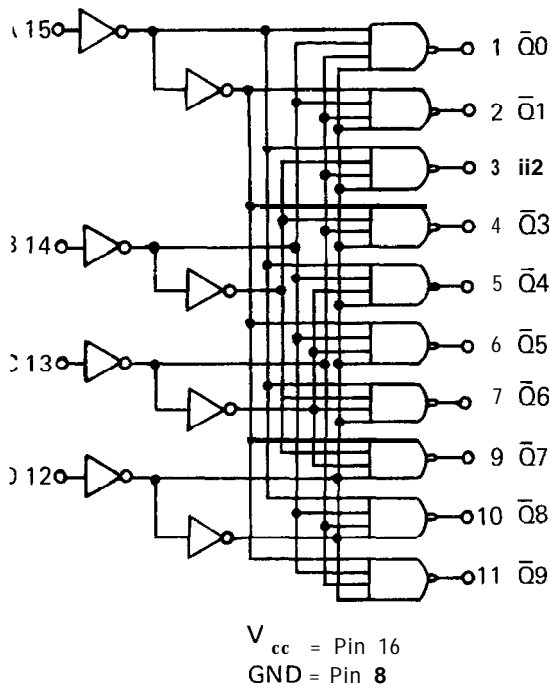


Fig. 7. Dual full adder.

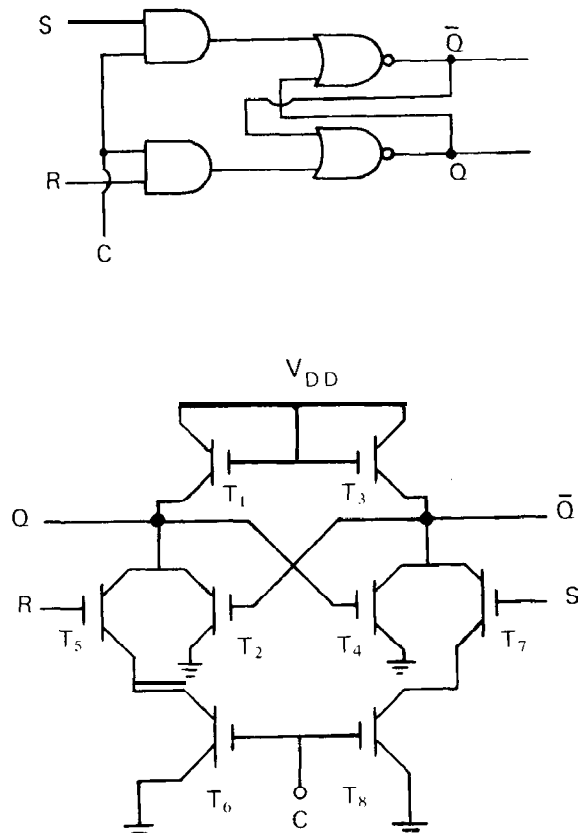


Inputs				Outputs									
D	C	B	A	\bar{Q}_9	\bar{Q}_8	\bar{Q}_7	\bar{Q}_6	\bar{Q}_5	\bar{Q}_4	\bar{Q}_3	\bar{Q}_2	\bar{Q}_1	\bar{Q}_0
0	0	0	0	1	1	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1	1	0	1
0	0	1	0	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	1	1	1	1	1	0	1	1	1
0	1	0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	1	1	1	1	1	0	1	1	1	1	1
0	1	1	0	1	1	1	0	1	1	1	1	1	1
0	1	1	1	1	0	1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

Fig. 8. BCD to decimal decoder.

greatly, and the output Q is raised to a high-voltage or 1 level. An examination of the truth table shows that the stored information will not be disturbed by either S or R when C stays low. When C is energized, the flip-flop can be set to 1 by energizing S , and reset to 0 by energizing R .

Shift Registers. Shift registers may be formed from a series of flip-flop circuits. A typical shift register is shown in Fig. 10, where the clock signal is



Truth Table

C	R	S	Q	a
1	0	1	1	0
1	1	0	0	1
1	0	0	Previous state	
0	0	1	"	"
0	1	0	"	"
0	0	0	"	"

Fig. 9. The logic layout, the truth table, and MOSFET implementation of a flip-flop.

connected to all the flip-flops. At each clock pulse, one bit of information will be written into the flip-flop FF1 from inputs A and B . At the next clock pulse, this bit will move to the flip-flop FF2 while a new bit is being written into FF1. Bidirectional shifting is possible when some additional control

COMPUTER CIRCUITRY

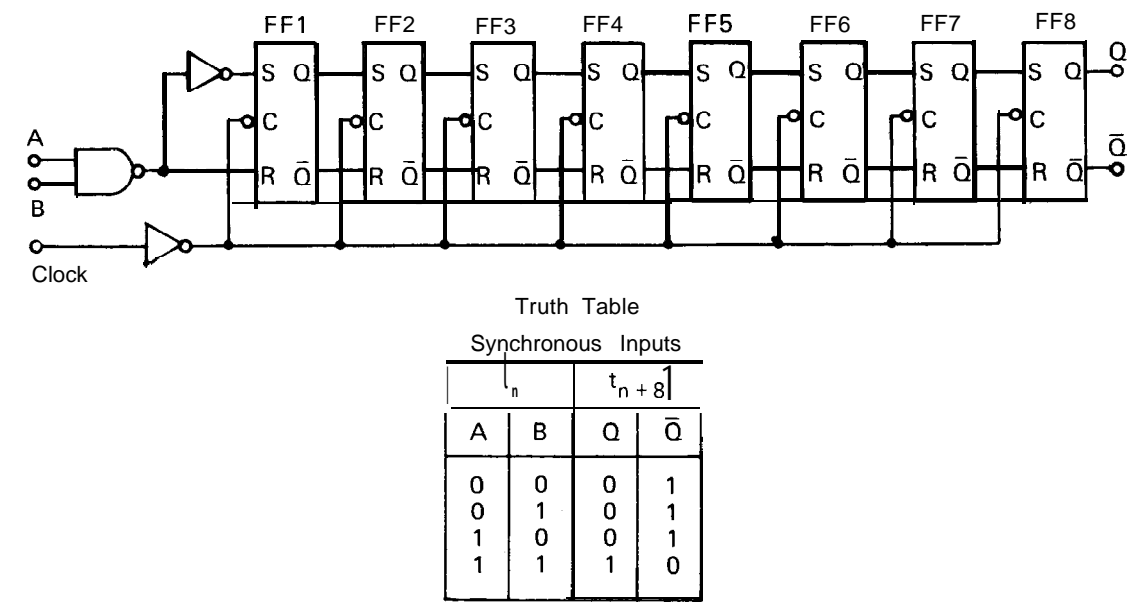


Fig. 10. An eight-bit shift register.

gates are added. Shift registers are especially useful when multiplication or division is performed in the arithmetic unit.

Future Computer Circuitry. The definition of a fourth-generation computer has not yet been well defined. One thing being generally agreed upon is that it will probably use large-scale integration (LSI) extensively. The logic circuit configuration in the fourth-generation LSI could be basically the same as those used in the third-generation computer. The LSI circuits could be either in bipolar or in MOSFET technology. As mentioned before, bipolar LSI will outperform MOSFET LSI in speed, but suffer from less density and higher power dissipation.

Through LSI, the number of chips and modules in a system can be reduced greatly. Besides cost reduction and reliability improvement, performance can be improved from reduced wire delay. However, LSI also creates such problems as (1) sizable design and initial tooling cost, (2) decreased yield with increased chip size, (3) excessive heat generation, (4) more complex debugging and testing, and (5) the limitation of the number of input/output (I/O) connection terminals.

In small-scale integration (SSI) and medium-scale integration (MSI), with circuit count roughly

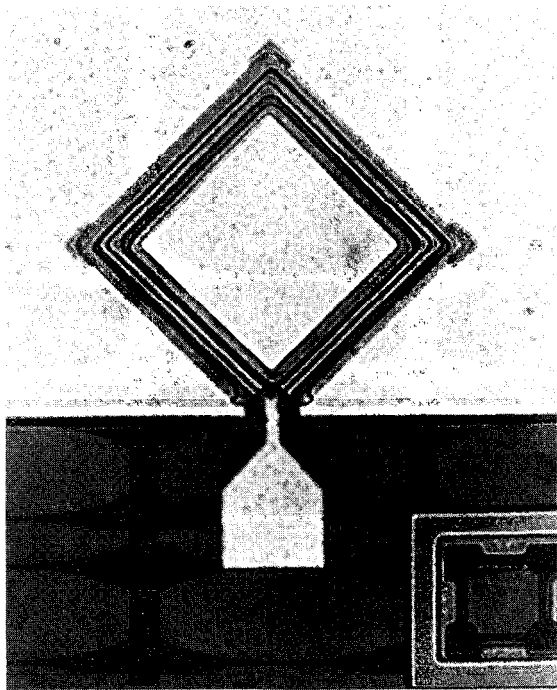


Fig. 11. Photomicrograph of a silicon field-effect transistor, about 100μ on a side and 1μ wide. (Courtesy of IBM.)

less than 10 and 100, respectively, air cooling is generally adequate for heat removal. In LSI, designers may be forced to choose between low-power circuits or a sophisticated cooling system. In SSI, and even in MSI, many nodes are accessible for debugging and testing. In LSI, however, the number of accessible nodes is greatly reduced, and testing can become quite complicated. When chip size is doubled to accommodate more circuits, the peripheral area, which accommodates the I/O connection terminals, is increased only 40%. This problem may become a limiting factor in LSI logic circuit chip design.

LSI logic circuit chips can be roughly divided into four groups:

1. Custom chips designed for a specific single application.
2. Master chips having a given number of circuits and I/O connection terminals, and designers have some flexibility in interconnecting some of or all the circuits to perform certain functions, such as the universal logic circuits.
3. Array chips featuring, generally, READ ONLY memory arrays to perform certain specific functions.
4. Functional chips that perform predesigned functions such as those offered by manufacturers in their catalog.

Most LSI chips are presently being used in electronic calculators or minicomputers. For example, an eight-bit parallel central processing unit has been placed on a single chip marketed by Intel. An arithmetic logic unit, an accumulator-register-counter stack, an instruction decoding and control unit, a timing unit, and an I/O unit are all built into this single chip.

Another type of LSI chip that may have important impact on future computer circuitry has so-called *dynamic logic*. In all the logic circuits discussed previously, information (logical 1 or 0) is represented by a steady-state voltage level. This level is usually maintained by a steady-state current that causes power dissipation and generates heat. In dynamic logic, logic states are represented by charging or discharging a capacitor, without steady-state current. Power consumption will be greatly reduced. Since dynamic logic does not require resistors and since high-ohmage resistors demand a large silicon area, the circuit density of dynamic logic in LSI is substantially higher than that of other logics. Dynamic logic usually performs logical functions through four-phase clocks and is slower than other logics in general. Four-Phase Systems, Inc., has

adopted dynamic logic in their IV/70 systems.

The future LST will probably evolve from the present LSI. Two possible trends of the future LSI may be seen in the example of Intel processor chips: (1) LSI will contain logic, memory, and control in a single chip to perform as a complete system; and (2) LSI will be built as a collection of proven functions to ease the testing problem.

REFERENCES

1962. Huskey, H. D., and G. A. Korn. *Computer Handbook*. New York: McGraw-Hill, chap. 20.
1967. Crawford, R. H. *MOSFET in Circuit Design*. New York: McGraw-Hill, chap. 1.
1967. Lo, A. W. *Introduction to Digital Electronics*. Reading Mass.: Addison-Wesley, chaps. 1 and 2.
1970. Kohavi, Z. *Switching and Finite Automata Theory*. New York: McGraw-Hill.
1972. Torng, H. C. *Switching Circuits, Theory and Logic*. Reading, Mass.: Addison-Wesley.

S. S. YAU AND I. T. Ho

COMPUTER GENERATIONS. *See* GENERATIONS, COMPUTER.

COMPUTER GRAPHICS

For articles on related subjects see **COMPUTER-AIDED DESIGN; CURSOR; DATA STRUCTURES; INPUT-OUTPUT DEVICES; JOYSTICK; LIGHTPEN; PICTURES, BASIC STRUCTURE; RAND TABLET; TERMINALS; and TIME SHARING**,

For articles on related terms see **BUFFER; FILES; MODELS; PERT/CPM; and RING**.

Computer graphics (graphics) may be defined as the input, construction, storage, retrieval, manipulation, alteration, and analysis of pictorial data. Computer graphics in general includes both off-line *input* of drawings and photographs via scanners, digitizers, or pattern-recognition devices, and *output* of drawings on paper or (micro) film via plotters and film recorders. *Interactive* graphics is a term used to emphasize man-machine dialog, which takes place in real-time using an on-line display console with manual input (interaction) devices (Fig. 1).

Among such input devices are the alphanumeric and function keyboards for typing text and activating pre-programmed subroutines, respectively,



Fig. 1. A Vector General graphics display console showing a **lightpen** pointing to a picture on the display screen, the function keyboard (lower left), the alphanumeric keyboard (lower center), joystick (right center above alphanumeric keyboard) and potentiometer controls (lower right). The Display Processing Unit, which is the hardware controlling the display, is under the table.

and the **lightpen** and data tablet for identifying and entering graphic information by means of pointing and drawing.

The scope of this survey article is restricted to interactive **graphics** (called "graphics" in what follows), and therefore excludes scene analysis and pattern recognition, image processing and enhancement, computer animation, etc., which are covered elsewhere in this encyclopedia. Newnan and Sproull (1973) discuss the technology surveyed here in far greater detail, and a good overview of applications and current trends is available (IEEE, 1974).

For various technological and historical reasons, most of today's graphics concerns line drawings (so-called wire-frame representations) of two- and three-dimensional abstractions such as electronic and mechanical circuits; structural components of buildings, cars, ships, and planes; chemical diagrams; functional plots of mathematical formulas; and flowcharts. In addition to line-drawing graphics, we are now beginning to see an

increase in interest in on-line manipulation of solid pictures with gray scale, color, and hidden line/surface representations of three-dimensional scenes (Sutherland, Sproull, and Schumacker, 1974).

HISTORICAL OVERVIEW. Line-drawing graphics started with the two display consoles on Whirlwind I, one maintaining a user screen and the other feeding a computer-controlled camera as a precursor of today's film recorders. The Sage tactical air defense system in 1955 had a multiconsole, multiuser display configuration with human feedback entering the system via lightpens that had pointing capability, toggle switches, and alphanumeric and function keyboards. Ivan Sutherland and his associates at the M.I.T. Lincoln Laboratories introduced and popularized most of the fundamental notions of graphics (still in use today) with their cleverly named, seminal SKETCHPAD system (Sutherland, 1963) on the TX-2 computer. Since the early 1960s there has been commercial development, with DEC, IBM, and (later) Tektronix supplying most of the hardware, and big industry such as General Motors and Lockheed as the pioneers in designing large-scale systems software and applications programs for the new hardware.

Today, after more than ten years, however, the field is still in its infancy, with many unfulfilled promises. Graphics as the appealing "window into the computer" has been oversold on the one hand, and has been forced to cope with many legitimate but somewhat unforeseen problems on the other hand. Among these problems are very expensive, nonstandardized display hardware with inadequate, nonstandardized software; insufficient realization of, and accounting for, the cost of central computer hardware and software resources to support the display terminal and its application programs; and the difficulty and cost of implementing sophisticated large-scale applications programs (typically of the CAD variety; see Prince, 1971), very few of which have turned out to be cost-effective.

Only recently have we begun to see the long desired emphasis on simple, straightforward, and cost-effective applications via widespread use of truly low-cost graphics terminals. These are typically storage tubes or refreshed displays driven by small minicomputers costing less than \$10,000 and programmed via a user-oriented subroutine package (typically embedded in Fortran). Indeed, **device-independent** graphic subroutine packages and languages are finally emerging to allow the user/pro-

grammer to concentrate on his application rather than on the peculiarities of his display or input devices. Similarly, the gap between "soft copy" on the display and "hard copy" prints is being closed by display-connected printout devices or plotters driven from the same program that builds pictures for the display. Given the much greater availability of reasonably priced hardware and easily used software, the long awaited era of man-machine symbiosis may finally have arrived.

APPLICATION AREAS. The following list, although admittedly far from comprehensive, gives at least an idea of the areas of use to which graphics has already been put.

- Verification drawings of numerical control tapes
- Weather maps
- Con tour maps
- Exploration maps for petroleum and mining
- Ship, aircraft, missile, and satellite course plotting
- Cartography, including oceanographic charts
- PERT network drawings
- Flight test and engine performance graphs
- Telemetry data plotting
- Highway cut and fill
- Research and engineering data reduction
- Temperature and pressure drawing
- Fourier analysis
- Antenna scatter display
- Optical ray tracing
- Calibration curves
- Mathematical studies function analysis
- Kinetic analysis
- Route layout simulation
- Reservoir sizing
- Power spectrum displays
- Quality control displays
- Oil production maps
- Cockpit and aircraft landing visibility studies
- Wave research drawings
- Pattern layout (clothing, metal)
- Layout drawings, printed and integrated circuits
- Layout drawings, petroleum and chemical processes
- Pole, line, and distribution drawings, electrical utilities
- Computer animation in science and entertainment
- Electrical, mechanical, structural, and civil engineering drawings
- Drafting and schematic and dimensioned drawings

- Subdivision and construction layouts
- Computer-aided design systems
- Models of human organs and physiological systems
- Architectural and planning designs
- Computer-aided instruction via graphical output and simulation

CLASSES OF GRAPHICS APPLICATION PROGRAMS.

The areas listed above can be categorized in a variety of ways. An obvious one is based on the type of picture generated: for example, whether two- or three-dimensional, or whether portraying an abstract or a real entity (such as a four-dimensional mathematical object or an idealized electronic circuit versus a perspective drawing of a house). A categorization that is less drawing-oriented and more programming-oriented divides the application areas into three reasonably distinct ones in a spectrum, based on the amount of man/machine interaction that they exhibit.

Interactive plotting describes probably three-quarters of graphics application programs. (Note: The term "application program" denotes the totality of the software system, including the graphics programs that produce the pictures, the interrupt handlers, the analysis routines, the data base routines, etc., as described in the next section). The display console is used simply to "browse" through output of computational processes, typically in the form of a graph prepared by the host computer. Little interaction is required except for the real-time display of successive frames on operator command, and simple "menu selection" (i.e., via labeled function key or **lightpen** identification of a command name displayed in a control area on the screen) to direct the browsing or further computation. Such applications as computer-assisted instruction and command and control operations fit into this category of primarily predefined pictures.

Design drafting describes the much more demanding and elaborate preparation of complex schematics and blueprints, typically those of industry. In these applications, an operator constructs a highly detailed drawing on line, using a variety of interaction devices and programming techniques. Facilities are required for replicating basic figures, achieving exact size and placement of components, making lines of specified length, width, or angle to previously defined lines, satisfying varying geometric and topological constraints between components of the drawing, etc.

One primary difference between these first two classes of graphics lies in the amount of effort the

COMPUTER GRAPHICS

operator contributes, with interactive design drafting requiring far more responsibility for the eventual result. In interactive plotting, the computation is of central importance and the drawing is typically secondary. A second difference is that design drawings tend to have structure, i.e., to be hierarchies or networks (interconnected assemblies) of mechanical or electrical components. These components must be transformed (translated, rotated, scaled, projected in standard engineering views, etc.) and edited (inserted, deleted, reconnected, and reconfigured) at the console.

If, in addition to nontrivial layout, the applications program involves significant computation on the picture and its components (transient analysis or stress analysis, for example), we speak of the third and most complex category, that of *interactive design*. In addition to a pictorial datum base, or data structure, that defines where all the picture components fit on the picture and also specifies their geometric characteristics, an *applications datum base* is now needed to describe the electrical, mechanical, and other properties of the components in a form suitable for access and manipulation by the analysis program. This datum base must naturally also be editable and accessible by the interactive user.

Relatively long (multisecond) responses from the host computer may suffice for interactive plotting, since the user may look at one complete display (frame) for quite a few seconds before requesting the next or recomputing to produce the next. Such delays are intolerable in design drafting and interactive design, however, because the user may spend the entire console session manipulating and altering perhaps only a few pictures in a cyclic "design, view-results, redesign" process. Furthermore, human factors play an essential role in making *graphics-based interactive design* a palatable substitute for pencil-and-paper methodology. For example, being able to zoom smoothly in and out on a large drawing and display the detail part in a user-defined window is a necessary feature for overcoming the limitations of a small, low-resolution console screen (see the section on Windowing). Another human-factor criterion to be considered is that the program must be constructed in such a way that a novice user should not be overwhelmed by too many details when he is still in the learning phase; this is only one aspect of "user-friendliness."

Review of Graphics Technology. The purpose of this section is to describe in very general terms the various hardware and software components of a graphics system. Terms and nomenclature

are introduced by example.

HARDWARE. Fig. 2 symbolizes the major hardware components with which the applications programmer and user deal directly or indirectly. (Note: The term "user" may denote either the application programmer who writes the graphic programs or the end user, who may be the noncomputer-oriented problem solver or draftsman. To avoid confusion, user in this article will mean end user.) A user's only contact with the configuration is confined to the display surface for viewing the picture (the output function) and to the manual input devices for creating and altering the picture and controlling the analysis program (the input function). All other hardware should ideally be of no concern to him.

For purposes of discussion, the display console is assumed to be attached via an I/O channel (as is any other peripheral) to a computer in which the graphics program is executed. In the display console, a buffer stores the set of display commands (called the "display file," display program, or buffer program) that defines the picture. This display file is directly executed by the *display controller* (also called display processor or display processing unit), in the same way that an ordinary program is executed by a CPU.

The display processing unit (DPU) decodes each command and causes the numeric description of the position of points and lines on the display surface to be transformed into analog movements of the electron beam in a cathode-ray tube (CRT), similar to that in a home television receiver. Phosphors in the screen surface emit light when stimulated by this moving electron beam. The phosphor's light output decays typically in tens of milliseconds, so the DPU on most displays must loop through the display file 30 to 60 times per second in order to restimulate the phosphor and provide a flicker-free image. This periodic refreshing of the screen is quite similar to what is done in a TV monitor by a **60-frame-per-second** interlaced TV signal. If too much information is contained in the display file, the beam cannot move around in the time available for each frame, and the image will flicker.

The display file, or DPU program, as described above is oriented primarily toward specifying output of pictures on the screen. Where is the specification of the handling of user input? The typical DPU simply passes interrupts from programmed function keys, lightpens, etc., to a low level CPU interrupt-handling module in the operating system, which in turn passes control to an applications program to

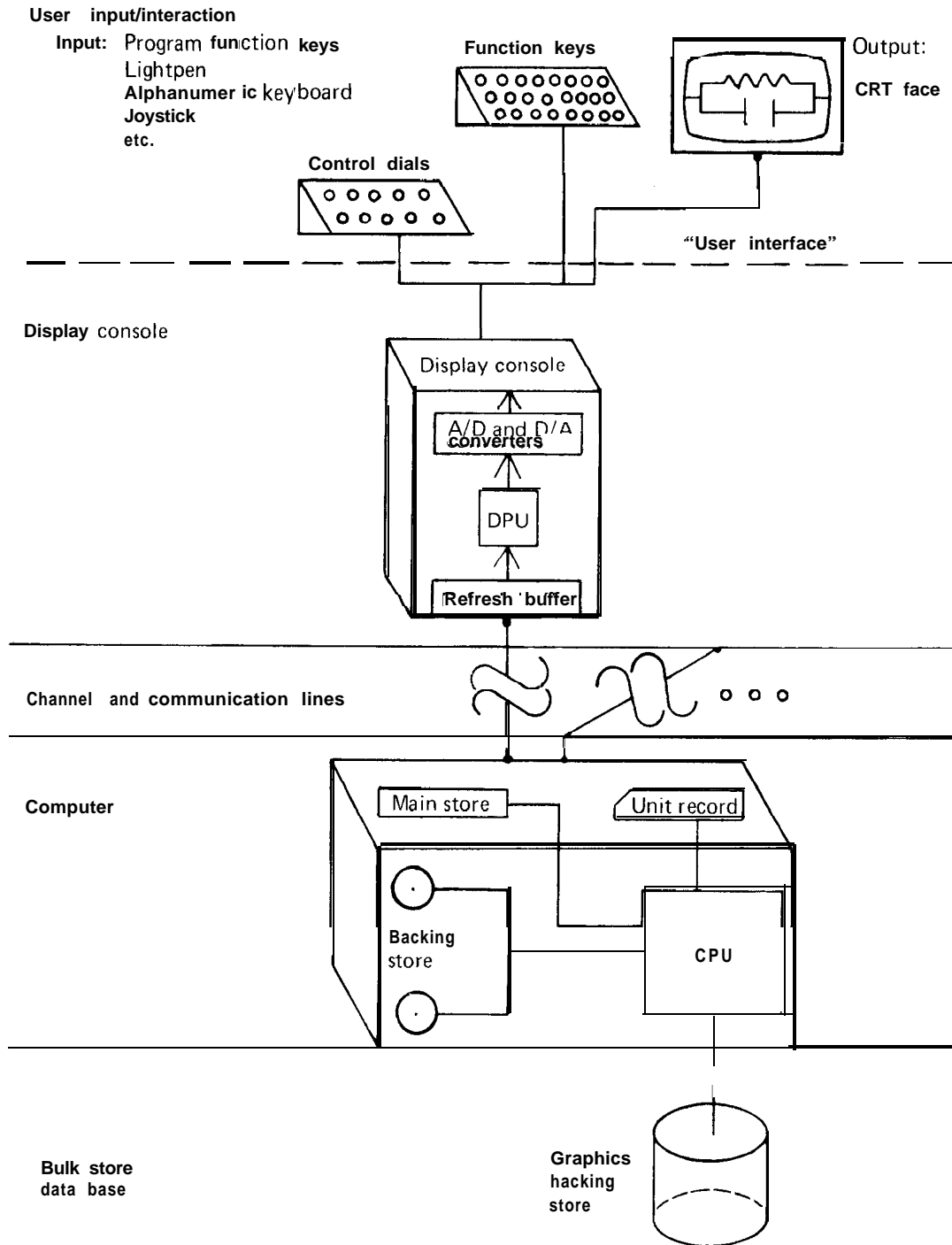


Fig. 2. Hardware view of configuration.

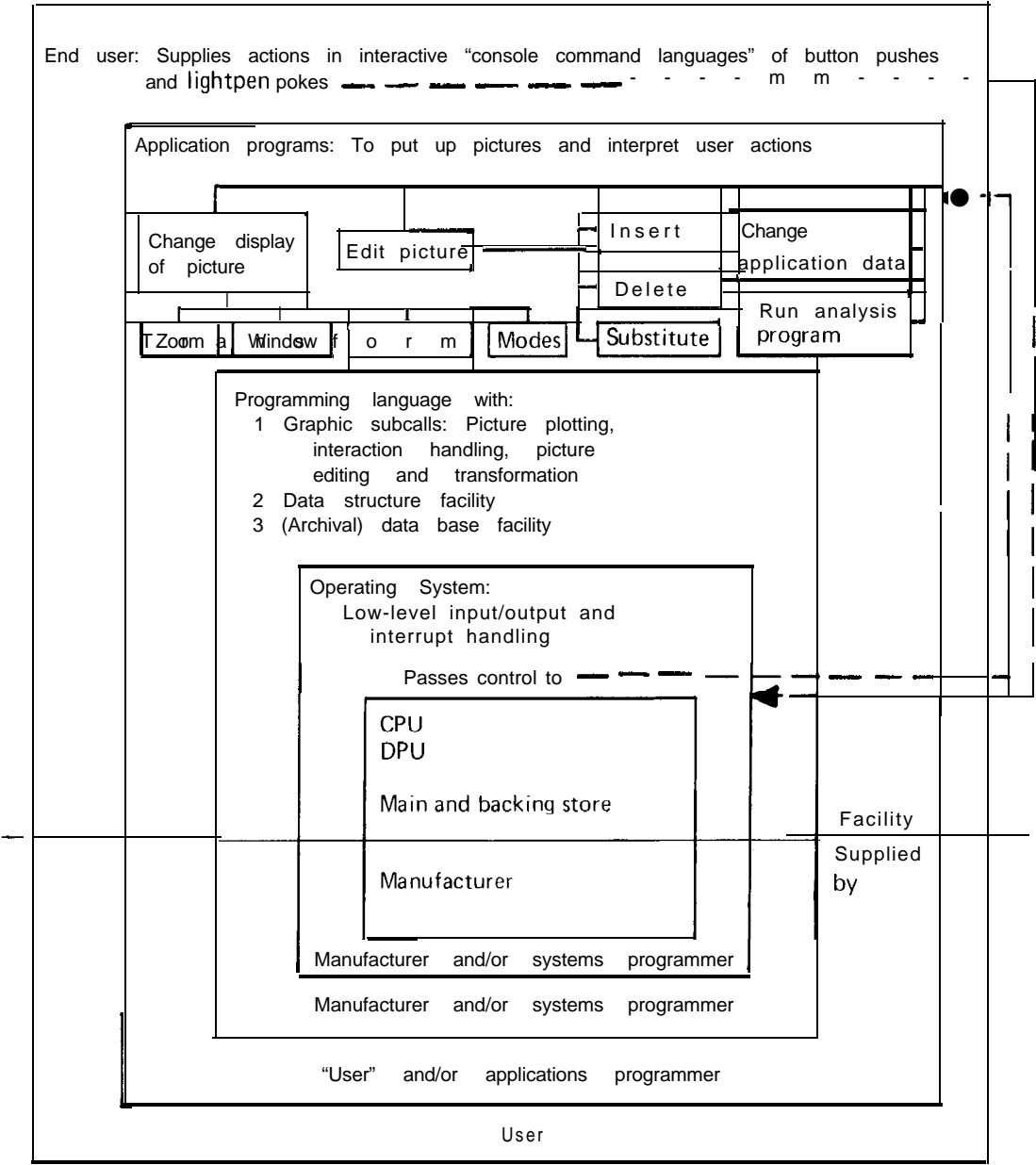


Fig. 3. Layers of software.

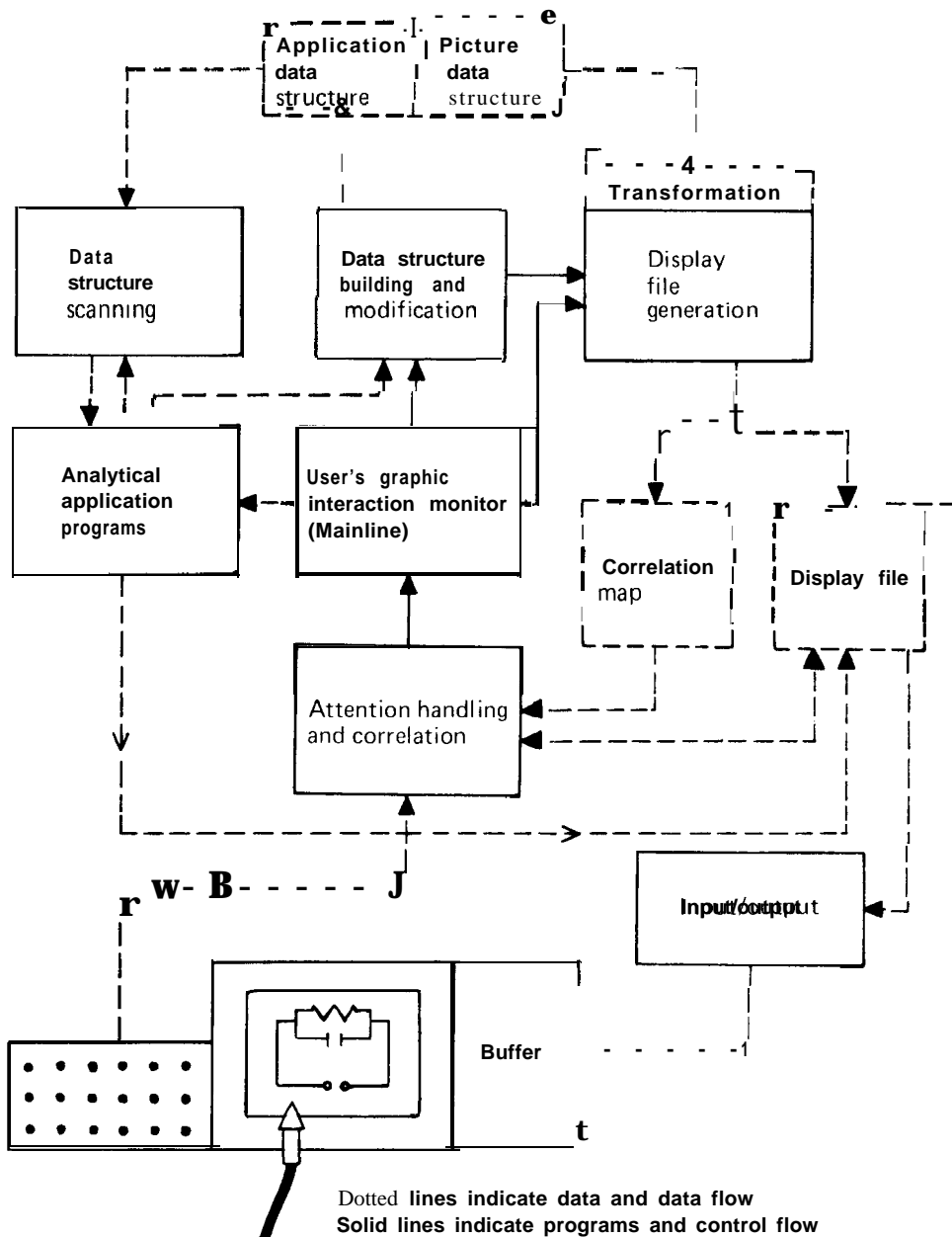


Fig. 4. Interrelationship among main software elements and data structure.

service that *attention* at a logical level.

SOFTWARE. The software point of view is shown in Fig. 3 as a series of shells or layers progressing from the level that the user deals with directly down to the inner core surrounding the bare hardware. While facilities and their formats may differ from installation to installation (e.g., types of features supported in the subroutine package or graphics language), most systems do conform to this type of hierarchy of levels. Naturally, the various levels do communicate with each other, but mostly without the user's direct involvement. The sequence of these intercommunications is discussed below.

SCENARIO OF A TYPICAL APPLICATION. The following scenario introduces additional terms and concepts by describing the interaction between man and machine as well as the underlying interaction between the various modules of the graphics program. Different organizations are possible for different application programs, or for different hardware, software, and data structure configurations; the scheme shown, however, is quite typical of a large class of interactive design applications.

The central routine in Fig. 4, the graphic interaction monitor, is little more than a dispatching table that channels requests made by the user via graphic attentions to the appropriate program modules. In fact, the entire system may be modeled by a state graph in which transitions from states are determined by user-generated inputs that result in actions such as the execution of routines, output of feedback messages to the user, and transitions to internal states (Newman, 1968).

Pictures are represented in a two-segment data structure (see next section) containing (1) pictorial data giving geometric and topological information that describes what the picture *looks* like on the screen [e.g., the electronic circuit in Fig. 5(a)], and (2) applications data defining what the picture means. Notions of syntax and semantics of pictures are sometimes used to distinguish these two structures. Picture meaning is defined in terms of the applications (analysis) programs that calculate the behavior of the entity symbolized by the picture on the screen, which is accomplished by manipulating the applications data structure. "Values" of components comprise the type of information usually stored in the application data structure. Naturally, some applications programs concentrate on geometry (e.g., mechanical structures), whereas others are virtually independent of geometry (e.g., topology, and not geometry, is important in network and circuit synthesis). In any case, picture display and picture analysis usually take place at different times

and usually alternate, as described below.

1. *Picture Display.* This involves reduction of the picture data structure to a **viewable** display file. Assume the entire applications program has been loaded and an initial prompting message has been displayed on the screen to orient the user. Some input device(s) (lightpen, alphanumeric or function keyboard) has been provided for use and the program waits for the user to generate an attention. Let us say he pushes a function key, generating an attention, which is mapped by the attention-handling module into the function key number. This number is in turn translated by the interaction monitor into a request for service—say, retrieval of a previously stored drawing. The retrieval program called by the monitor readies the keyboard for new input and adds a prompting message (such as, "please type in name of drawing") to the display file from which the console is refreshed.

When the user next transmits the typed-in name, the attention is again passed through the monitor to the retrieval program, which retrieves the named picture from the disk library and puts it in the in-core work area, passing control to the display-file generation program. This program next reduces the device independent picture data structure into a DPU-specific set of display commands constituting the display file, which is subsequently loaded into the buffer. The *correlation map* created at the same time is a mechanism for mapping **lightpen** "detects" on individual points and lines in the display file back to the pictorial data structure items to which they belong.

2. *Picture Modification.*

The Delete Process. Let us assume that the user wants to delete a subpicture on the screen. Again, he must activate the appropriate module by pushing the appropriate function key (or by menu selection through lightpenning a "light button"—the command name displayed as a **lightpen** sensitive character string on the screen), which is channeled by the monitor to the "delete" subroutine. This subroutine, which is part of the data structure building and modification module, next enables the **lightpen** and/or the typewriter keyboard to allow the user to identify the subpicture he wishes to delete. If the user points at (a line on) a resistor, the attention handling and correlation routine determines which pictorial data structure item has been identified, deletes the data structure representation of that particular resistor, and calls the display file generator to update the display file in place or to replace the previous display file in the buffer.

In order to prevent the user from inadvertently deleting the wrong part of the data structure (in case of incorrect specification or a change of mind), a properly human-factored program would not go through the above procedure until a "soft-delete" had been tried out by the user. This process updates the display file by inserting a branch around the display code of the item to be deleted; it then prompts the user to ask him if that result was the one desired. If he accepts (another pass through the attention handler and the monitor), the above data structure and display file update is done.

Note that in the typical cycle that takes place while the display is being refreshed, the monitor waits for the attention handler to report the occurrence of a user-instigated attention. When the attention is received, it passes control to a subroutine, which initiates a sequence of intelligence-gathering operations from the user. As soon as the user has specified sufficient information, his initially designated subroutine carries out its task and waits for approval. It is this *attention-driven dialog* that makes graphics programs quite different from ordinary batch programs (and from other interactive programs, for that matter).

An additional feature of the delete process is that it tends to cause a "deletion ripple" in the data structure. For example, if an endpoint of a line is deleted, the user probably also expects the line to disappear, since it is no longer well defined. The line, in turn, might have been used in a constraint relation with other picture parts, or it might be labeled, causing additional updating of the data structure and display file. If a node in a network were to be deleted, the user might similarly want to delete all components attached to that node. The utility of the "soft-delete" procedure with its option for allowing the effects of an operation to be undone ("reverted") should be obvious at this point.

The Analysis Process. In Fig. 4 the display file generation step is purposely drawn symmetric to data structure scanning. Both scan the data structure in order to extract information, the former being primarily geometric pictorial information, the latter being topological and applications data. As another typical scenario, therefore, assume that the user has finished construction of his picture. To invoke an analysis program, he pushes an appropriate function key. The monitor transfers control to the data structure scanning program, which examines the part of the data structure that contains parameters such as physical dimensions and properties necessary for analysis. The data values are extracted and passed to the analysis program, which may display feedback to

the user in the form of graphed results; it may also allow the user to identify elements of the picture with his pointing device in order to get a display of appropriate parameters at the point indicated. At this time, a cycle of redesign usually takes place in which the user modifies the picture in order to submit it for re-analysis.

Note the preponderance of data structure operations; very little output is displayed on the console without first involving data structure scanning, manipulation, and reduction. While this phenomenon is not necessarily unique to graphics, three key characteristics of graphics programs do make them differ. First, and most obviously, the hardware driven by programs is distinct in terms of its capabilities and instruction repertoire. Second, in addition to the normal data types such as characters and numbers, programs deal with points, lines, subpictures, and other geometric notions. Finally, along with other interactive programs, graphics programs spend most of their time waiting for the user to drive them; flow of control is more accurately modeled by the state graph than by straight-line, branched, or even looped-flow graphs used for batch programs.

Despite these differences, most of the code in graphics programs does not, in fact, deal with producing pictures or fielding interrupts. In common with most other (interactive) programs, the bulk of graphics applications programs is concerned with command-language parsing and interpretation; data structure manipulation, computation, and analysis; space management for main and backing store, etc. It is therefore fair to say that in addition to the obvious differences, graphics programs require all the normal concerns that any other nontrivial program induces.

SUMMARY OF REQUIRED SOFTWARE. As shown in Fig. 3, software support required for writing graphics systems such as the one described above must include many facilities. Among these are those for describing the layout of pictures (picture-plotting software), decoding user input (attention-handling software) and subsequent picture transformations, alterations, and manipulations (picture-editing software). Additionally, the scenario showed the need for main-store data structure handling software and backing-store data base handling software. (Data structures and transformations are discussed in more detail in the last two sections.)

The exact nature of these facilities differs from installation to installation. The most common form of support is a graphic subroutine package, typically embedded in Fortran, which provides picture plot-

ting and some amount of transformational capability, and adequate attention handling. Graphical languages or extensions of existing languages such as Fortran or PL/I with graphical constructs have been noticeably less successful and are not commercially available.

Role of Data and Storage Structure in Graphics. Manipulating (as opposed to merely plotting) pictures requires a conceptualization, or modeling, of the picture, which goes beyond a literal point-and-line representation. The totality of geometric and topological data, including the hierarchy of picture parts and associations between them, constitute a model of the picture referred to above as the pictorial data structure. This machine-independent data structure must naturally be encoded in digital storage in a machine-specific storage structure. Thus, "the storage structure . . . is the image of the data structures in some computer memory" (D'Imperior, 1969). The storage structure should lend itself readily to displaying the picture, manipulating and transforming the picture at the console, and causing analysis routines to be applied to (parts of) the picture.

Sutherland's interactive SKETCHPAD established full ring structures as canonical storage structures for "master/instance" definitions of picture/subpicture hierarchies (see "Classes of Structures" below). Since that time (1963), hardware suppliers

have for the most part ignored the data-structure/storage-structure aspect of graphics because the problems of implementing a data/storage structure package are far more difficult than those of providing graph-plotting subroutines or even graphic attention-handling subroutines. Only the latter have been provided, typically within a Fortran environment in which it is not natural and often inefficient for the user to implement the data/storage structures himself (primarily due to the lack of pointer and structure variables in Fortran). Most installations, having realized that the largest part of a comprehensive interactive design program deals not with picture plotting but with data structure manipulation, have been forced to implement their own data structure support, primarily in assembly language (Gray, 1967). The SLIP subroutine package for Fortran (Weizenbaum, 1963) has been used occasionally, and a few installations have embedded graphics support in PL/I, which does provide most basic data structure facilities. Other installations (frequently university graphics laboratories) have used derivatives of Bell Lab's Low-Level Linked List Language (L⁶; Knowlton, 1964) to provide a reasonable compromise between efficiency of code produced and machine independence of the source language.

CLASSES OF DATA STRUCTURES. The simplest type of storage structure required is that for displaying primitive pictures made up of consecutive

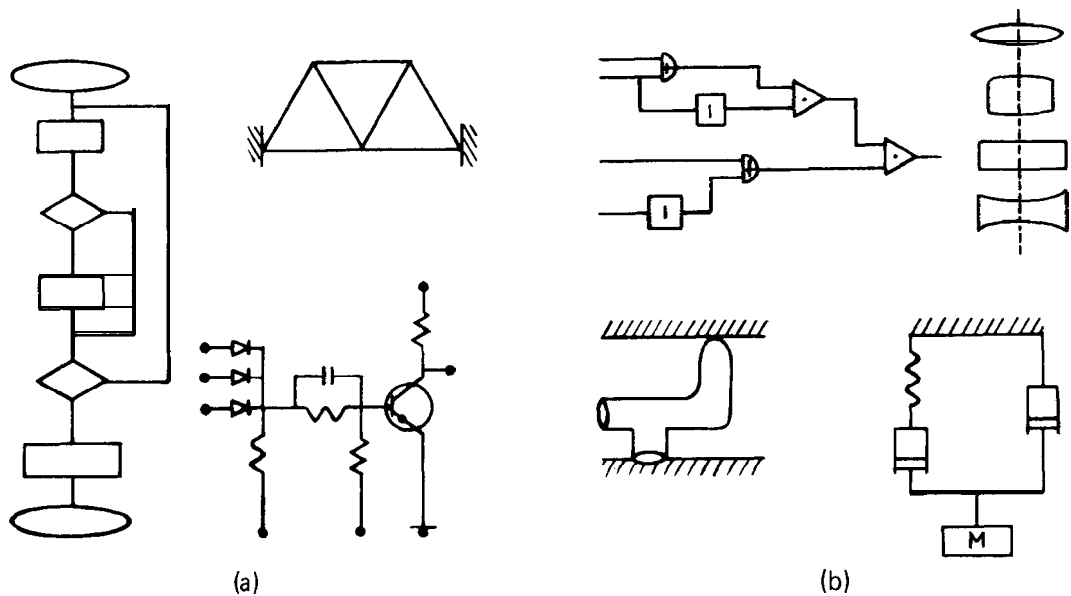


Fig. 5. (a) NOR circuit; (b) other network graph examples.

points and lines, say, for plotting purposes. In this case, a pair of arrays, one for points and one for lines, is all that is required. Most pictures have more structure, requiring minimally some facility for collecting logically related *picture atoms* (points, lines, characters) in a group and naming this group so it can be manipulated as a unit (to be deleted, moved on the screen, made sensitive or insensitive to the *lightpen*, etc.). Ordinary data graphs and many mechanical and structural drawings, for example, require no additional facilities.

Sutherland handled a very common class of more complex pictures that could be called "network graphs" (Fig. 5). These are two- or three-dimensional configurations in which discrete pictures are connected to others in a network, and are typically decomposable into a hierarchy of lower-level subpictures.

In the typical "bottom-up" interactive construction of a subpicture hierarchy (e.g., electric circuit design), components are gathered into subassemblies, which in turn are used in higher-level assemblies; in the "top down" method (e.g., flow charting), loose (macro) descriptions are iteratively refined and expanded into subassemblies of more detailed (micro) boxes. The "NOR" circuit of Fig. 5(a), for example, can be constructed bottom up as a hierarchy, as shown in the "parts explosion" of Fig. 6. Practically, one would not build up such an electric circuit starting at the low level shown here for illustration purposes, but would start at the level of primitive electrical components such as resistors and transistors. Note that the tree form of the data

structure is not strictly accurate in that some of the nodes are duplicated,

The data structure use of a subpicture in a given picture closely parallels the programming use of a subroutine in a higher-level routine. The original data structure definition is called the "master," and its invocation is **known** as an "instance" of the master. The "subroutine" parameters, encoded with each instance portion of the storage structure for the total calling picture, consist of the geometric parameters that determine position, orientation, and scale of the subpicture within the calling picture. Often, such parameters are collected in a matrix called the "transformation" matrix (see concluding section). As with program subroutines, the advantage of using subpictures with transformation matrices is the space saved by not duplicating the definition.

A PRACTICAL EXAMPLE. Figs. 7 through 9 represent the data and storage structures of a very large (greater than 700K bytes) applications program in engineering design. The 3DPDP (Strauss, 1969), a three-dimensional piping-design program, allows the user to lay out a three-dimensional piping configuration in on-line stereo mode or in any one of the three "engineering views." The configuration may then be analyzed for flexibility under thermal loading.

The data structure is that of a typical network: The circles (Fig. 7) called "tangent intersections" (TINs), represent the nodes of the network, and the straight-line piping segments (tangents), the "bends," and the "anchors" form the components of

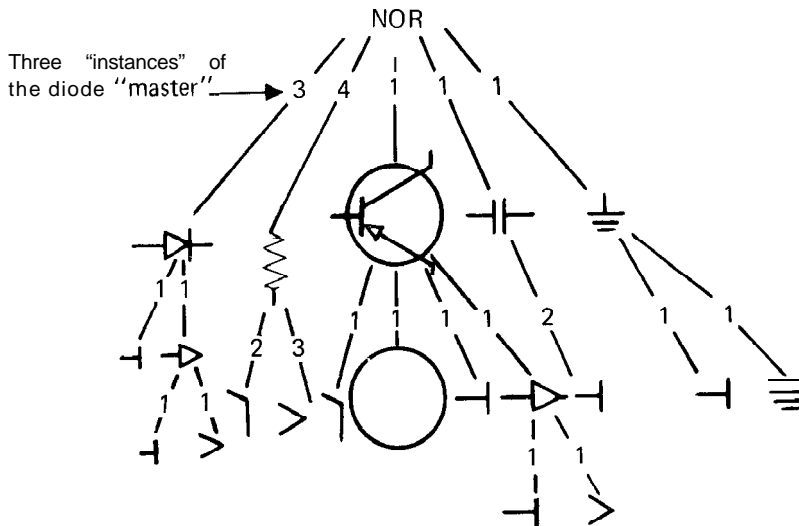


Fig. 6. Hierarchical representation of NOR.

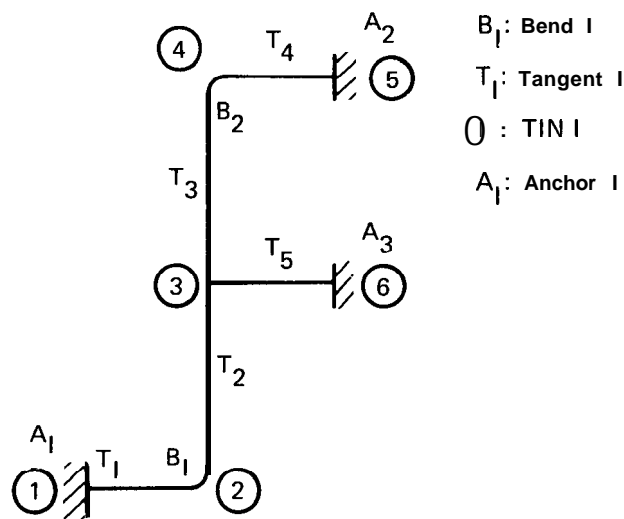
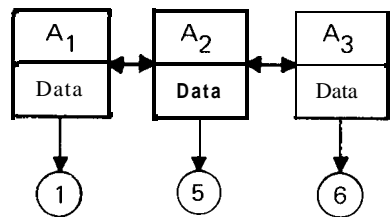
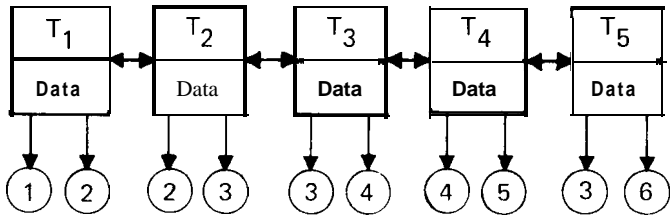


Fig. 7. Tangent intersections.

Anchor List



Tangent List



Bond List

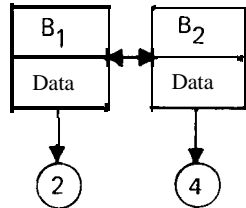


Fig. 8. Component lists.

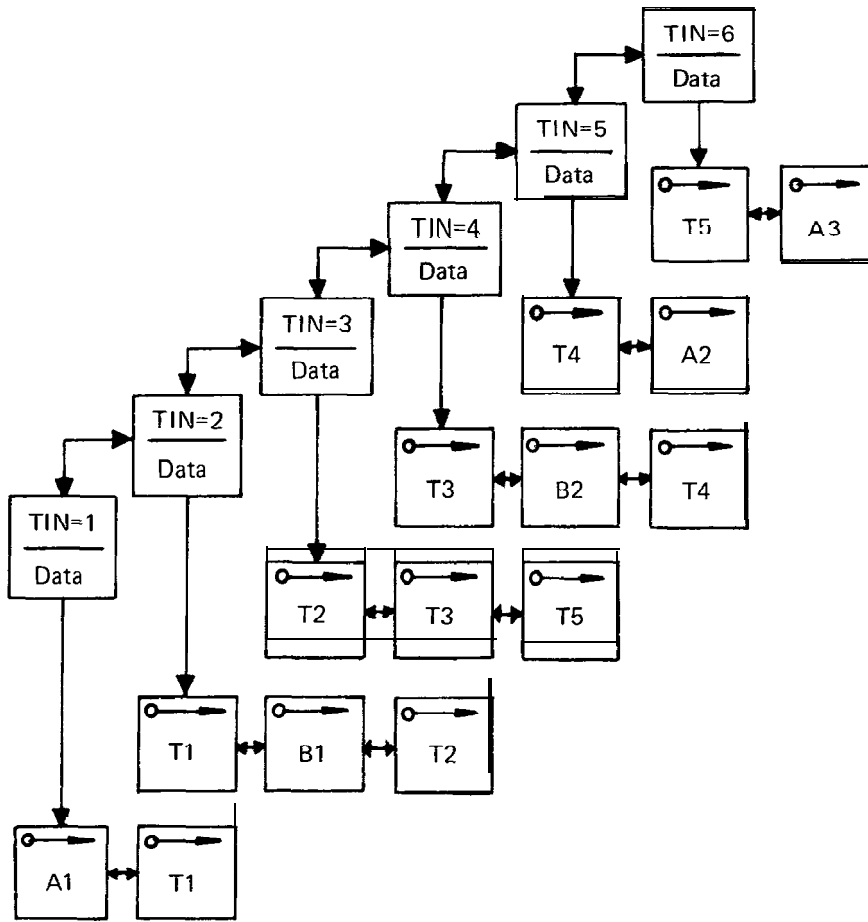


Fig. 9. TIN list.

the network. For purposes of display and for the analysis program, the designer found it useful to maintain lists of each of the three types of components (Fig. 8). The storage structure is a two-way linked list of blocks that contain an identifier, a string of physical properties, and finally one or two pointers to the nodes of the network to which the particular component is attached. The topology and geometry of the network are captured in the tangent intersection list (TIN list, Fig. 9), which is again encoded as a two-way linked list of blocks. Each block contains an identifier for the node, a string of physical properties, and a pointer to a two-way linked sublist. Each block on the sublist consists of the name of a component attached to the particular node, with a pointer indicating its location.

As an example of a data structure process, consider the method by which a user obtains results from the flexibility analysis program pertaining to

the components attached to a given anchor. Assume that anchor A3 has been pointed at by the user with a lightpen. The A3 block on the anchor list (Fig. 8) is accessed via the correlation map, a pointer to node 6 is obtained, and the node 6 block on the TIN list (Fig. 9) is retrieved. Its sublist shows that tangent T5 is attached, and the proper values may be obtained from the analysis program. (Another method might be to have the analysis program store an explicit network connection matrix.)

Note that this simplified storage structure does not allow for picture/subpicture hierarchies, nor for cementing sections of large piping diagrams together. While the first facility is easily implemented by enriching the data structure with a tree hierarchy similar to that of Fig. 6, the second requires a hardware or software paging facility (van Dam, 1972).

Picture Manipulation

GEOMETRIC TRANSFORMATIONS AND REAL-TIME DYNAMICS. As mentioned in the preceding section, graphics becomes especially powerful through the ability to compose complicated pictures from other previously drawn pictures suitably transformed to fit into higher-level pictures. While it is not feasible to discuss all possible geometric transformations in a survey such as this, an indication can be given of their inherent simplicity by showing a two-dimensional example of the most used ones: translation, rotation, and scaling.

First, as shown in Fig. 6, any picture can be recursively defined in terms of its component points, lines, character strings, and (sub) pictures, each suitably sized and positioned on the coordinate system of that picture. We would like to find formulations which allow us to transform a point, and given that ability, transform the other types of components by simple extension. The transformations below allow just that: A line is transformed simply by transforming its endpoints; a character (string) is moved by moving the center or left bottom corner of the first character (rotating and scaling may be tricky without sophisticated hardware); and an instance of a subpicture is transformed by transforming it relative to the local origin with respect to which it was drawn and which serves as its "handle."

Many display manufacturers supply special hardware to carry out the transformations summarized below. Some even provide the ability to combine the basic transformations with windowing and perspective mapping, thereby allowing very complicated picture manipulations to proceed in real time as the user twists knobs, dials, and joysticks, and "flies around" or "inside" his picture world. If hardware is not available for real-time dynamics, software simulation may still provide useful and interesting effects.

First, to move a point, add x - and y -translation factors to the x - and y -components of the coordinate pair describing the point. To move a line, move both endpoints by the same factor. Thus, in Fig. 10, to move (x_0, y_0) by $(\Delta x, \Delta y)$:

$$\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Second, to rotate a point about the center (origin) of the coordinate axes (the z -axis, in effect), rotate the vector with the beginning point at the origin and the endpoint at the desired point. Thus, in

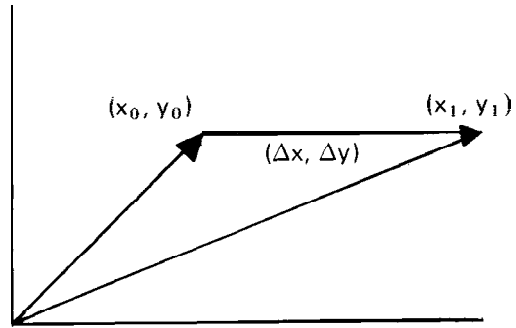


Fig. 10. Translation of a point.

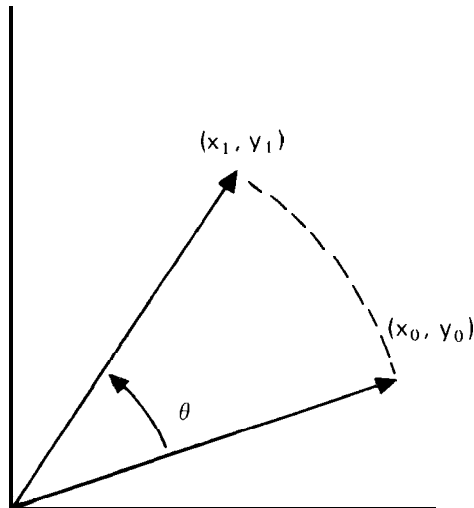


Fig. 11. Rotation of a point about origin.

Fig. 11, to rotate (x_0, y_0) about the origin by θ degrees, use the matrix multiplication.

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

If we were to rotate the point about any other center of rotation, or were to rotate a line about either of its endpoints, we would have a slightly more difficult problem. Since the rotation formula allows us to rotate only about the origin, we could rotate about an arbitrary point only if we first moved it down to the origin. This is simply a well-known mathematical trick for reducing a given problem to a previously solved one. Thus, to rotate line L_1 (Fig. 12) about P_1 , we translate P_1 to the origin, then apply the rotation matrix, and then put P_1 back where it belongs. In a similar manner we can rotate an entire subpicture

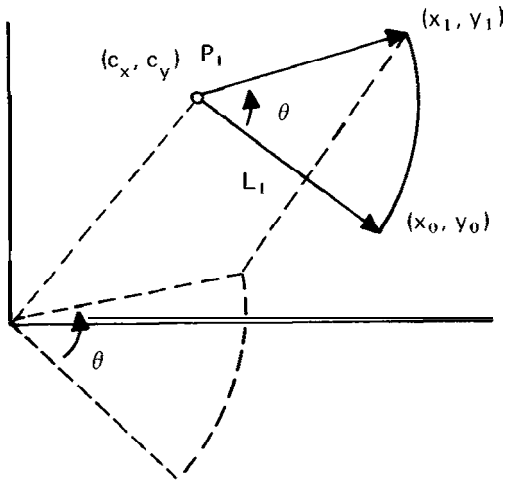


Fig. 12. Rotation about arbitrary centers.

instance about its local origin (rather than the picture origin) by translating, rotating, and "untranslating" all the individual picture components. Naturally, all these calculations may be simplified by "solving" these matrix and vector operations beforehand, i.e., reducing them to simple equations for the endpoints. Thus, in Fig 12, rotating L_1 about P_1 by θ degrees:

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \left(\begin{bmatrix} x_0 \\ y_0 \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \end{bmatrix} \right) + \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

To scale a line (scaling a point doesn't really make sense), we multiply the x - and y -components of its endpoints by x - and y -scale factors. Thus, in Fig. 13, scaling by different amounts in x and y (A , B):

$$\begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

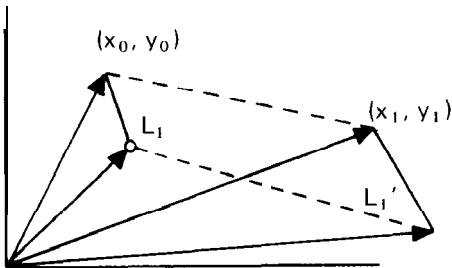


Fig. 13. Scaling.

Note that, in general, this will have the effect of moving the line as well. If we want a given point on the line to stay in place, a compensating translation must be applied. As with rotation, an entire subpicture instance (i.e., all its components) may be scaled about its local origin, by moving the local origin to the picture origin, scaling as desired, and moving it back. If scaling about axes inclined with respect to the picture axes is desired, the local axes should be translated down to the origin rotated to be parallel to the picture axes, scaled, unrotated, and then unrotated (Fig. 14).

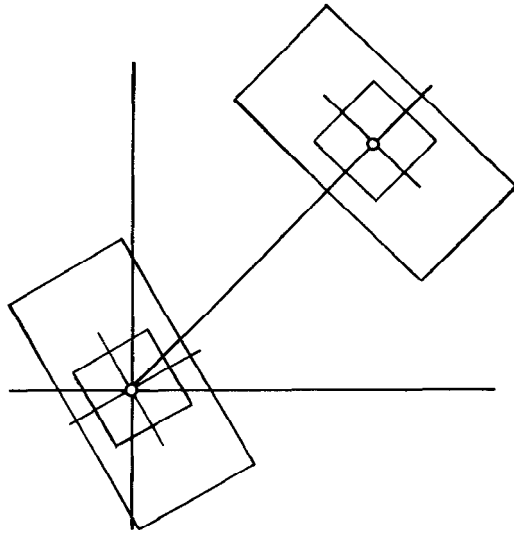


Fig. 14. Scaling about arbitrary axes.

WINDOWING. Since large drawings of the type typically found in engineering applications cannot fit in their entirety on small display screens, we can either compress them to fit-and thereby obscure details and induce clutter-or we can display only a portion of the total drawing. The portion to be displayed is usually indicated by the user by placing a rectangular window on the compressed version of the drawing on the screen, and the hardware or software will then "clip" ("scissor") off any points, lines, and characters that fall outside the window. The operation may be performed repeatedly to achieve any desired degree of magnification.

A two-dimensional window is usually defined by a maximum and minimum value for x and y , or by a center of the window and maximum relative x - and y -values. Simple subtractions or comparisons suffice to find out whether or not a point is in view.

COMPUTER GRAPHICS

For lines, the algorithm should allow any part of a line within the window to be displayed. If both endpoints lie in view, the line may be trivially accepted. If one of the endpoints lies in view, finding the other point is easy enough, and at least a portion of the line lies in the window. However, if both points are out of view, further tests must be made, since the line could be wholly outside or could cut the window. One method for deciding which case applies is to solve analytically for points of intersection of the line containing the line segment with the lines forming the edges of the window (Fig. 15).

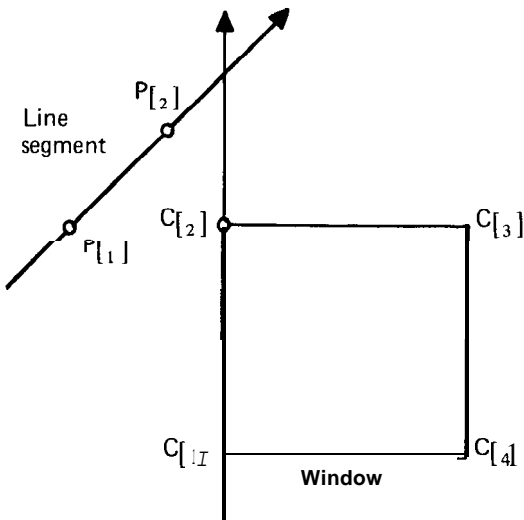


Fig. 15. Windowing.

Using notation from linear algebra, the equation of a line can be expressed in parametric form such that points of the form

$$X = tX[1] + (1 - t)X[2]$$

for real t are on the infinite line through the points $X[1]$ and $X[2]$. When t is restricted to the interval $[0,1]$, the point is on the directed-line segment from $X[2]$ to $X[1]$. The problem can therefore be stated as finding parameters t and s such that

$$tP[1] + (1 - t)P[2] = sC[1] + (1 - s)C[2]$$

If both t and s are between 0 and 1, then the point of intersection is both: between $P[1]$ and $P[2]$ (so it actually lies on the line segment), and between $C[1]$ and $C[2]$ (so it is in view in the window).

As soon as one point to be displayed has been found, the line segment may be treated as two separate line segments, each of which has one endpoint in view. One of these segments may be rejected completely, as shown in Fig. 16.

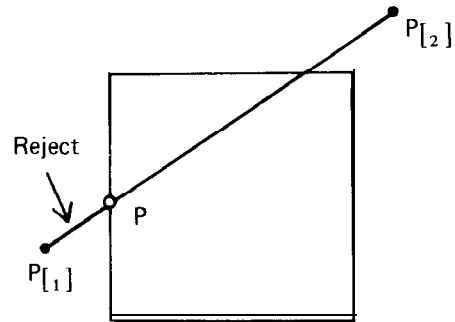


Fig. 16. Windowing: rejection of line segment.

To augment this straightforward analytical solution, special-purpose hardware has been built using various digital or analog methods for performing the window operation in real-time (Sproul and Sutherland, 1968). Both in hardware and software implementations, much effort is expended in accepting or rejecting the trivial cases (elements wholly inside or outside the window).

ENHANCING THE ILLUSION OF REALITY. The computer-generated scenes shown thus far in this article are not meant to be realistic; they are diagrammatic and symbolic in nature and are simple to draw. Presenting a realistic reproduction of a complex three-dimensional scene is several orders of magnitude more complex. First, line drawings themselves are often inadequate: We do not view real objects as sequences of lines but rather as various surfaces, some of which are connected one to another. Surfaces have color, texture, and light reflectance and transmittance properties. A solid surface close to the viewer can hide more distant surfaces (or portions thereof) from view. Objects of equal size appear to become smaller the farther they are from the viewer (perspective). Depending on where the source of illumination is, objects may cast shadows on other objects, making them appear darker than they are normally.

Considerable effort has been devoted to developing appropriate mathematical models and algorithms that take these various factors into account. The basic problem is the massive amount of computation required to transform a mathematical mod-

el into a picture. Current research efforts are thus directed as much toward improved algorithms as toward increased realism. A few of the faster algorithms to remove hidden lines and surfaces and do shading have been partially implemented in special-purpose hardware (Sutherland, Sproul, and Schumacker, 1974).

The applicability of this work to interactive computer graphics lies in the areas of design, simulation, and animation. It is easy to conceive of an automotive designer using pictures like Fig. 16 to view his current efforts from many directions. In simulation, we want to achieve effects such as the presentation of a realistic road scene in a driving simulator, or a realistic airport scene in a flight trainer. The making of animated movies for entertainment or for scientific, mathematical, and medical modeling is another area needing realism.

A Sampling of Other Configurations and Technologies. The display system organization shown in Fig. 2 is not the only one in use, but is probably the most widespread. Another kind of organization is the satellite graphics system. The display processor is connected to a small computer, which is in turn connected to the large host computer. The graphics system and application programs are distributed between the two computers, which may be separated by many miles. Motivations for such systems include placing the graphics terminal where the user is, not necessarily where the computer is, to provide fast response to simple user actions and to unburden the host CPU of some processing work (van Dam, 1974).

Another variation on the basic system organization is the use of a rectangular raster (TV) scan to present a picture. A line of a given length and orientation would not be drawn by a single continuous deflection of the CRT's beam, but by turning the beam on for each point of the sequential raster scan corresponding to a point on the line. This technique is attractive because TV technology is far less expensive than the random-scan deflection mechanism. The disadvantage of it is that an image must be broken down into a series of on/off commands, to be applied as the beam sweeps its scan down the screen. This is usually a time-consuming, relatively low resolution process, although a few special-purpose processors are able to do it as quickly as the image is displayed.

Raster-scan technology is especially applicable to many of the current hidden-line/hidden-surface algorithms. It also adapts nicely to conventional video-mixing techniques.

Other variations use the direct-view storage tube (DVST) and the plasma display panel. The DVST stores the electron beam drawn picture in a dielectric mesh in which the cathode-ray tube phosphor is embedded, and therefore obviates cyclical refreshing of the DPU from a stored display file. DVST consoles therefore can display an unlimited amount of information, but do not lend themselves to selective erasure of portions of the screen, nor are they as fast as ordinary cathode-ray tubes.

Plasma panels are solid-state matrices of individually *XY-addressable* picture elements (60 elements per inch is a typical resolution available today) that also exhibit memory. The panels are flat, can display color or superimposed static information from slides, and unlike cathode-ray tube devices, are easily batch-fabricated in various sizes.

Graphics Progress. Graphics has suffered in the past decade from a preoccupation with its fascinating and still rapidly evolving technology and hardware; too little attention has been paid to cost-effective, possibly even mundane, applications programs, and to making life easy for the ordinary user. Fortunately, this trend is changing, and we can look forward to some of the promises of graphics (such as ease of use, naturalness, and ready availability) finally becoming fulfilled. A cardinal rule, obvious but nonetheless often overlooked with unpleasant consequences, is that only simple things are simple to do (e.g., graph output of computational processes). Sophisticated interactive design systems take considerable amounts of people, hardware, and software resources, and should therefore not be underestimated. Despite the visual wizardry, there is no magic in computer graphics.

REFERENCES

- 1963. Sutherland, I. E. "SKETCHPAD," *Proceedings of A FIPS 1963 SJCC*, Vol. 23.
- 1963. Weizenbaum, J. "Symmetric List Processor," *Communications of the ACM*, Vol. 6, No. 9.
- 1964. Knowlton, K. C. "A Programmer's Description of L⁶," *Communications of the ACM*, Vol. 9, No. 8.
- 1967. Gray, J. C. "Compound Data Structure for Computer-Aided Design-A Survey," *Proceedings 22nd ACM National Conference*.
- 1968. Newman, W. M. "A System for Interactive Graphical Programming," *Proceedings of AFIPS 1968 SJCC*, Vol. 32.

COMPUTER-MANAGED INSTRUCTION (CMI)

1968. Sproull, R. F., and I. E. Sutherland, "A Clipping Divider," *Proceedings of AFIPS 1968 FJCC*, Vol. 33-1.
1969. D'Imperio, M. "Data Structures and Their Representation in Storage," Annual Review in *Automatic Programming*, No. 5. New York: Pergamon.
1969. Strauss, C. M. "3DPDP—A Three-Dimensional Piping Design Program," Ph.D. Thesis, Brown University, Providence, R.I. (June).
1971. Prince, M. D. *Interactive Graphics for Computer-Aided Design*. Reading, Mass.: Addison-Wesley.
1972. van Dam, A. "Some Implementation Issues Relating to Data Structures for Interactive Graphics," *International Journal of Computer and Information Sciences* (August).
1973. Newman, W. M., and R. F. Sproull, *Principles of Interactive Computer Graphics*. New York: McGraw-Hill.
1974. Institute of Electronic and Electrical Engineers. "Special Issue on Computer Graphics," *IEEE Proceedings* (April).
1974. Resch, R. D., "Portfolio of Shaded Computer Images." Special Issue on Computer Graphics, *IEEE Proceedings* (April).
1974. Sutherland, I. E., R. F. Sproull, and R. Schumacker, "A Characterization of Ten Hidden-surface Algorithms," *ACM Computing Surveys* (March).
1974. van Dam, A., G. M. Stabler, and R. J. Harrington, "Intelligent Satellites for Interactive Graphics." Special Issue on Computer Graphics, *IEEE Proceedings* (April).

A. VAN DAM

COMPUTER-MANAGED INSTRUCTION (CMI)

For articles on related subjects see **COMPUTER-ASSISTED LEARNING AND TEACHING**; and **COMPUTER-ASSISTED INSTRUCTION**.

Computer-managed instruction (CMI) refers to the use of computer assistance in testing, diagnosing, prescribing, grading, and record keeping. Some writers prefer "computer-aided management of instruction" in order to emphasize computer assistance to

the human teacher or counselor, in contrast with management *by* the computer.

Computer assistance has been made available in many ways to those managing instruction, including aids for students managing their own instruction. The teacher of a large class finds assistance in scoring tests, keeping records, checking on which students need what kind of work, and **computing** grades. A manager of a self-instruction group uses the computer to obtain summary records showing where each student stands. A student or teacher may call upon the computer files and procedures to generate a test at random but according to set rules. The procedure may select from an item pool and plug in variations on standard question forms to obtain the specific test items so they appear fresh each time. Computer-based information systems are used by students and teachers to locate instructional materials in various media according to needs, interests, and the limitations of course time and instructional budget.

Major projects using the computer for assistance in the management of instruction are located in Pittsburgh (Learning Research and Development Center), Philadelphia (Research for Better Schools), Palo Alto (Project PLAN of the American Institute for Research), and the Medical School at Ohio State University (Columbus). Each successful program is based on a large amount of curricular materials, probably in modular form, and a convenient testing and record-handling system. The arguments for CMI instead of CAI include: lower cost of operation, since students spend less time at computer terminals; more flexibility in learning formats, since students are referred to materials in a variety of media and learning settings apart from the computer; lower cost of development, since existing materials can be used for instruction. CMI and CAI may be used together; the management aids associated with CMI can refer the student to selected exercises that are presented by the computer (CAI) as well as to many others that do not benefit from presentation in the computer medium.

REFERENCE

1971. Baker, Frank B. "Computer-Based Management Systems; A First Look," *Review of Educational Research*. [AERA] Vol. 41, No. 1. pp. 51-71.

K. L. ZINN

COMPUTER NETWORKS

For articles on related subjects see **ARPA NETWORK; COMMUNICATIONS AND COMPUTERS; COMPUTER SYSTEMS; COMPUTING ECONOMICS; ACQUISITION AND OPERATION; DATA COMMUNICATION NETWORKS; DATA COMMUNICATIONS; MULTIPLEXING; PACKET SWITCHING; TERMINALS; and TIME SHARING.**

For articles on related terms see **HARDWARE MONITORS; MINICOMPUTERS; and REMOTE JOB ENTRY.**

The term "computer networks" has been used to describe situations in which:

1. Geographically remote terminals and RJE stations are connected to a central computer.
2. Geographically remote smaller computers are used for minor editing tasks and to transfer input to magnetic tape and from magnetic tape to output printers and plotters. Magnetic tapes are transferred by post or messenger between smaller machines and a central computer,

3. A central computing unit has connections to smaller machines with specialized functions, which provide it with services such as storage (and associated file management) and communication facilities (such as message concentration); in the case of small machines used for graphics, the central unit is used for major computing tasks.

4. Independent major computing systems ("hosts"), possibly in addition to the above, communicate with one another, and share resources such as hardware, programs or data (Fig. 1).

Computer networks should not be confused with information networks, a term usually applied to systems for the sharing of library resources. However, it is clear that, particularly with the growth of computerized printing (which provides text in machine-readable form-e.g., on magnetic tape-as a byproduct), a growing source of loading for computer networks will be their use as information networks.

The fourth definition of computer networks -resource-sharing networks of machines of comparable power-is coming to be the more widely accepted one. As an indication of the recent growth of activity in this area, a bibliography published by

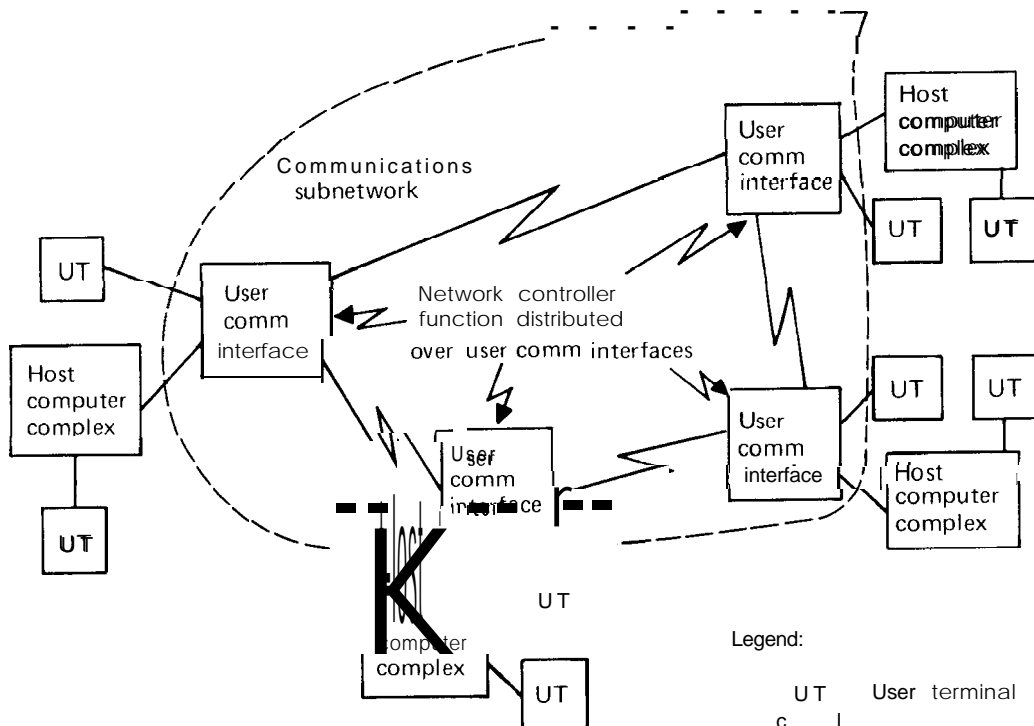


Fig. 1. Computer network. (Courtesy IEEE.)

COMPUTER NETWORKS

the National Bureau of Standards (Blanc et al. 1973) lists over 500 citations relating to resource-sharing networks, nearly all of which were developed in the past decade.

The rate at which computer networks are now proliferating throughout the world indicates that they are becoming a powerful force in both the public and private sectors, both nationally and internationally. This upsurge of activity, which is of recent origin, may be ascribed to three main technological trends :

1. The greatly increased reliability of computers, which makes possible the implementation of complex systems that would have been unworkable a decade ago.
2. The availability of low-priced minicomputers suitable for carrying out most of the functions required to operate a network, with a minimum of change to the operating systems of the major connected computers (these operating systems are large, complex, and not designed to provide the quick interrupts needed for communications work).
3. Major changes in communications technology, which are reflected in a corresponding reduction of communication costs.

In terms of their broad end-use, computer networks, as can some other branches of human activity, can be categorized as:

1. Monolithic empires, constructed for a single organization and an explicit purpose (such as an airline reservation system).
2. Alliances of several approximately equal partners (such as the North Carolina Triangle Universities Computing Center (TUCC) with three partners).
3. Free enterprise resource marketing facilities (such as TYMNET, a network operated since 1969 by Tymshare Inc.).
4. Facilities introduced by legislation (such as the state network in New Jersey) to consolidate state computing facilities.
5. Facilities constructed to acquire experience in a new experimental technique (ARPANET, sponsored by the U.S. Department of Defense Advanced Research Projects Agency (ARPA), was initially in this category).

Network Components. For a fuller description of various technical aspects of the design of networks, the reader is referred to Abramson and

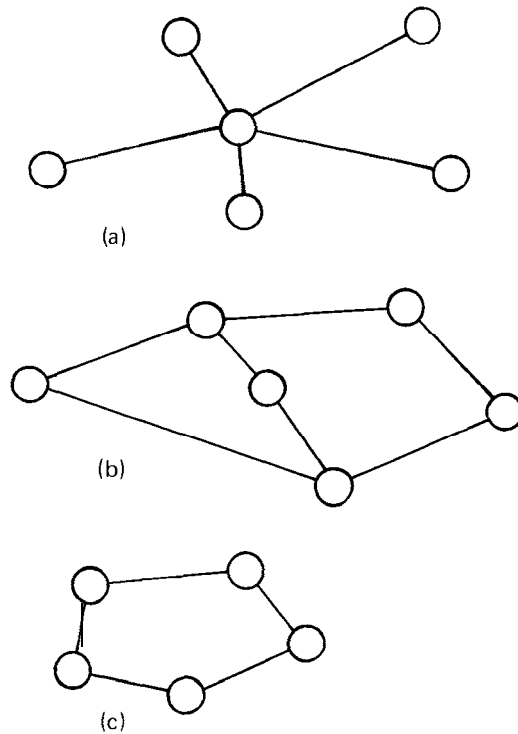


Fig. 2. Alternate network configurations: (a) star; (b) distributed; (c) ring. (Courtesy IEEE.)

Kuo (1973). For an introduction to communications techniques, Martin (1969) should be consulted.

Computer networks may be viewed (Fig. 2) as being composed of nodes, with circuits, channels, or links connecting them. A node may vary from a small amount of fixed hardware logic to one of the major connected computers. Nodes may be used to support network connectivity as store-and-forward computers (i.e., to receive and store messages, and to dispatch them along one of several different routes), as concentrators (e.g., take input characters from a number of slow terminals and assemble them into blocks), and as attachment points for major computer systems.

The extent of a network is to some extent arbitrary. A network may, for example, be considered to include the programs in a host computer which are needed to communicate with it, but does not include attached terminals. The nodes and linking circuits of ARPANET are referred to as the "communications subnet."

The details of physical channels (circuits or links)-which may be lines, microwave links, radio links, cable TV installations, or satellites-are of

little consequence from the point of view of computer networks. The relevant parameters of a channel are its maximum data rate, its error characteristics, and its directional limitations. The setup characteristics are also important information if a point-to-point circuit is not always dedicated to a network. This information may include the signaling mechanism and delays for circuit setup and breaking.

Commonly available speeds vary from 60 to 300 bps (bits per second), suitable for supporting a slow-speed terminal, through voice-grade line speeds of 2,000 to 4,800 bps (or higher) to 5×10^4 bps, the speed currently used by ARPANET. Higher speeds are available if required-systems to carry several gigabits per second (a gigabit equals 10^9 bits, a volume equivalent of the *Encyclopaedia Britannica*) -are under development.

Requirements for data transmission arising in connection with computer networks are typically burst-oriented, with transmission from terminals or computers for a short period at a specified rate, and long time gaps with no transmission at all. Standard communications techniques (frequency division multiplexing, FDM; or time division multiplexing, TDM) make better use of available bandwidths by allocating smaller bandwidths or time slots to individual subchannels, but make no use of the burst characteristics of the data. If a channel is connected to one subchannel only when that subchannel is active, an address being added to indicate the source (available from timing considerations with FDM and TDM), more efficient use will be made of the channel; this approach is referred to as asynchronous time-division multiplexing (ATDM).

Since it would not be economic for every node (or its equivalent) to be connected to every other node with which communication may be desired, transmission is usually routed through a number of intermediate nodes. In a typical case, these nodes are minicomputers and the connecting links are leased lines. Messages with suitable header information and error checks are passed from one node to another on a store-and-forward basis, the route being chosen according to loading or fault conditions by the minicomputers or by separate machines with control and monitoring responsibilities. Because messages are variable in length, problems arise in selecting sizes for buffer storage, and so it is usual to break a message into fixed length segments called "packets." These packets are transmitted on a store-and-forward basis and, with some networks, may go to their destination by different routes if changes occur in the loading of the links. Typical packet sizes are between 1,000 and 2,000 bits for text, with an

additional 100 bits for header information.

Packets are checked after passage through each link (an acknowledgment being returned to the sending node if correct), are reassembled in correct order at the destination, and passed on to the relevant process in the host computer to which they are addressed.

When nodes are allowed to compete for a channel, there may be a clash; in this case the check sums will not tally, no acknowledgment will be sent, and retransmission may be arranged to occur after a delay, which is different for each node so that a second clash is unlikely. The ALOHA system at the University of Hawaii uses a single radio channel in this way. The efficiency of this method can be raised considerably by introducing a reservation system in which a number of time slots may be reserved, one of these being subdivided into smaller slots that convey information about reservation requests to all user stations so that each station will know the position of the next free time slot.

At the time of this writing, there is still considerable controversy as to the extent to which packet-switching techniques should replace message-switching techniques or the more conventional circuit-switching techniques, in which a path is established from host to host by a dialing operation or its equivalent. For example, the British Post Office, although it has currently embarked on the construction of a network to evaluate the practicability of packet switching, has stated on several occasions that evidence of the practicability and viability of the packet mode of working is not sufficient to justify planning it as the main basis for any public service. Clearly, the "best" solution depends on the mix of message lengths involved, and the position is complicated by the trend toward digital transmission for general telecommunication usage.

Network Configurations. With the simplest network [Fig. 2(a)], a star network, all communication between the points of the star must take place through a central node. If this node is inoperable, the network cannot function.

Networks with alternative routings between nodes are referred to as "distributed" networks [Fig. 2(b)]. The reliability of a network may be assessed (Abramson and Kuo, 1973) by determining the number of nodes or links that must be inoperative before the network becomes disconnected, i.e., before there ceases to be at least one path between any sender and any receiver (apart from the removed nodes).

COMPUTER NETWORKS

A further design consideration is the maximum delay in a network. This delay (which is, for example, a half-second for individual packets in the case of the ARPA network) may vary according to the end-use of the data and the loading. However, once it has been specified, it is an important parameter: A network should be designed so as to achieve it for stated loadings at a minimum cost.

One structure [Fig. 2(c)] that has particular advantages is the loop or ring network structure (Farber and Larson, 1972). This structure lends itself to a TDM technique in which a node with a message for another node places packets in empty slots as they appear, and copies messages addressed to it as they are passed around the ring. When a message originating at a node is returned to it, it is checked to insure that it has been received and that it has not been corrupted, and is then replaced by a vacant slot.

NETWORK MONITORING. A detailed knowledge of network traffic is essential for network planning and operation, and node computers *should* contain suitable monitoring programs to make this possible. Hardware monitors operating under computer control can supplement these software monitors. The General Electric network monitor system (Wedburn and Hauschild, 1974) is relevant, as is recent work at the University of Waterloo (Morgan et al., 1974).

Economics. It is difficult to make definite statements now about the relative economics of networks except for those constructed for special purposes (such as airline reservation systems). However a number of commercial systems (e.g., TYMNET, Honeywell, and Cybernet) marketing facilities under a single management are clearly commercially viable.

Roberts (1974) has given some interesting figures concerning the economics of ARPANET. He advances arguments to show that work carried out through the network in 1973 would have cost about three times as much had equivalent local computing power been used, and that the difference more than offsets the annual cost of the network, even though at the time the network was only about 20% loaded.

Roberts draws some interesting conclusions about long-term trends. He points out that computing costs are being reduced by a factor of 10 every five years, as are satellite communication costs, whereas conventional land-line costs (which control the cost of transmission from ground stations to city centers) are reducing at the much lower rate of a factor of 10 every 22 years. These line costs would

represent 80% of network costs by 1980 were it not for the anticipated introduction of higher transmission frequencies, which are likely to eliminate the need for land lines by making possible the direct satellite communication to city centers. (This situation already applies with experimental stations that form part of the Pacific Educational Computer Network being constructed under the aegis of the University of Hawaii. There, ground stations each cost as little as several thousands of dollars.)

Management Problems. To make resources available through a computer network, a user must

1. Make arrangements for accounts at each host computer system.
2. Know the control language for each host.
3. Learn the peculiarities of network protocols (i.e., network management information provided in headers such as destinations and lengths of messages) as implemented by each host.
4. Determine what help facilities (if any) exist at each host and how to use them.
5. Determine who at each site can assist with systems problems and how to establish contact with him.
6. Determine how to get data and/or programs to and from the serving site.
7. Learn how to use the resources of the remote site.

It is small wonder that the average user is deterred from making the best use of resources available to him. Clearly, there is a strong case for brokers who know what is available and will help the potential user, acting as retailers of computing power available to them through the network on a wholesale basis and as liaison links for strengthening help facilities obtainable through the network itself. REX, a resource location and acquisition service offered by the Mitre Corporation (Benoit and Graf-Webster, 1974), is an example of this type of advisory service. It concerns the resources of ARPANET, and is offered through the network itself, permitting terminal users to converse interactively with it.

In the United States, the problem of funding network management has been facilitated by a decision made in November 1973 to approve the establishment of commercial "value added" communication networks (Doll, 1974). Operators of these networks would obtain raw bandwidth from common carriers such as AT&T, and "repackage" it, using minicomputers for leasing to the ultimate user.

PROTOCOLS AND STANDARDS. Protocols within a network must be standard so that nodes can function in a uniform manner; much of this information, because it is concerned only with the mechanics of packaging, will be supplied by the communications computers. Moreover, some additional information may be required by individual processes, and this may vary from one installation to another, particularly with a heterogeneous network (i.e., one that has as hosts different computers that are not compatible with one another).

Transmitting information from one network to another with a different protocol and packet length presents special problems, and the use of a special internetwork processor (a "gateway" machine) for the express purpose of reformatting messages and changing to new protocols has been proposed. An IFIP Technical Committee Working Group (WG6.1) is currently formulating guidelines for internetwork protocols, on which future standards may be based.

The field of data transmission has so far been the subject of over 50 standards and international recommendations laid down by major United States and international standards organizations (ANSI, EIA, ISO and CCITT) (10).

Some Typical Networks. The following examples of general-purpose networks are selected as illustrative rather than exhaustive. ARPANET is described elsewhere in this encyclopedia.

MERIT, an educational computer network, links machines at Michigan State University (CDC 6500), University of Michigan (IBM 360/67), and Wayne State University (IBM 360/67) through small communications processors. Bandwidths of links can be varied dynamically by providing each communications computer with four modems and call-up facilities.

CSIRONET is a network constructed by CSIRO in Australia, primarily to provide a research computing service for its own use. Its principal machine is a CDC 7600, with a Cyber 172 as a front-end processor. It is a star network, with terminals connected to six PDP11s, which serve as computers at nodes. At four of these centers, CDC 3200s provide RJE facilities.

TYMNET is a distributed network operated for profit by a major time-sharing company, Tymshare Inc. Currently, it has over 10,000 interactive users in 70 cities throughout the United States and Europe. It contains 100

communications nodes (Varian 620s), operating in a "store and forward" mode.

CYCLADES is a general-purpose distributed computer network constructed under the sponsorship of the French government. With 16 host computers, it uses a distributed five-node packet-switching communications sub-network CIGALE, with MITRA-15 minicomputers as nodes.

EPSS, the experimental packet-switching service being constructed by the British Post Office, will have nodes in three major cities with 48Kbps duplicated links between them. The nodes are being designed to handle 60 character-at-a-time inputs and 62 packet-at-a-time inputs at various speeds, and consist of similar modular units (Ferranti ARGUS 700Es), each handling about a third of the connected lines at each node and interconnected through fast store-to-store links, and provided with similar backup arrangements.

REFERENCES

1969. Martin, J. *Telecommunications and the Computer*. Englewood Cliffs, N.J.: Prentice-Hall.
1972. Farber, D. J., and K. C. Larson. "The System Architecture of the Distributed Computer System," *Proc. of the Symposium on Computer Networks*, the Polytechnic Institute of Brooklyn (April).
1973. Abramson, N., and F. F. Kuo (Eds.). *Computer-Communications Networks*. Englewood Cliffs, N.J.: Prentice-Hall.
1973. Blanc, R. P., I. W. Cotton, T. N. Pyke, Jr., and S. W. Watkins. *Annotated Bibliography of the Literature on Resource Sharing Computer Networks*, NBS Publication 384 (September).
1974. Benoit, J. W., and Erika Graf-Webster. "REX-A Resource Location and Acquisition Service for the ARPA Network." Washington, D.C.: The Mitre Corporation (January).
1974. Doll, D. R. "Telecommunications Turbulence and the Computer Network Evolution," *Computer*, Vol. 7, No. 2, pp. 13-22.
1974. Morgan, D. E., W. Banks, W. Colvin, and D. Sutton. "A Performance Measurement System for Computer Networks," *Proc. of IFIP 74*. Amsterdam: North Holland Publishing Co., pp. 29-33.
1974. Roberts, L. G. "Data by the Packet," *Spectrum*, Vol. 11, No. 2, pp. 46-51.
1974. Schutz, G. C., and G. E. Clark. "Data Com-

COMPUTER SCIENCE

munications Standards," *Computer*, Vol. 7, No. 2, pp.32-41.

1974. Wedberg, G. H., and L. W. Hauschild. "The General Electric Network Monitor System," *Proc. of IFIP* 74. Amsterdam: North Holland Publishing Co.

J. M. BENNETT

COMPUTER SCIENCE

For articles on related subjects see **DATA PROCESSING**; **INFORMATION PROCESSING**; **INFORMATION SCIENCE**; and **SYMBOL MANIPULATION**.

For articles on related terms see **ALGORITHMS, ANALYSIS OF**; **ARTIFICIAL INTELLIGENCE**; **AUTOMATA THEORY**; **COMPUTER ARCHITECTURE**; **COMPUTER GRAPHICS**; **COMPUTER SYSTEMS**; **FORMAL LANGUAGES**; **INFORMATION RETRIEVAL**; **LOGIC DESIGN**; **NUMERICAL ANALYSIS**; **OPERATING SYSTEMS**; **PROGRAMMING LANGUAGES**; **SIMULATION**; and **UTILITY PROGRAM**.

Computer science is concerned with information processes, with the information structures and procedures that enter into representations of such processes, and with their implementation in information processing systems. It is also concerned with relationships between information processes and classes of tasks that give rise to them.

The Domain of Computer Science. Even though the domain of discourse in computer science includes both man-made and natural information processes, the main effort in the discipline is now directed to *man-made* processes and to information processing systems that are designed to achieve desired goals (i.e., machines). The reason for this lies in the phenomenal growth of the computer field, its rapid penetration into almost all aspects of contemporary life, and the resulting pressure to bring some order into what is being done in the field, to educate the people behind the computing machines and to provide intellectual guidance for new developments in computer designs and applications. Thus, the bulk of empirical material currently available to computer science consists of systems, processes, and operational experience that grew in the computer field during the past quarter-century.

Clearly, the empirical corpus in the science is not stationary. It is growing with new development in the computer field. Some of these developments are themselves stimulated by the ongoing activities in computer science.

The main objects of study in computer science today are the digital computer and the phenomena surrounding it. Work in the discipline is focused on the structure and operation of computer systems, on the principles that underlie their design and programming, on effective methods for their use in different classes of information processing tasks, and on theoretical characterizations of their properties and limitations. Also, a substantial effort is directed into explorations and experimentation with new computer systems and with new domains of intellectual activity where computers can be applied.

The central role of the digital computer in the discipline is due to its near-universality as an information processing machine. With enough memory capacity, a digital computer provides the basis for modeling any information processing system, provided the task to be performed by the system can be specified in some rigorous manner. If its specification is possible, then the task can be represented in the form of a program that can be stored in the computer memory. Thus, the stored program digital computer enables us to represent conveniently and implement (run) any information process. It provides a methodologically adequate, as well as a realistic, basis for the exploration and study of a great variety of concepts, schemes, and techniques of information processing.

There exist in nature information processes that are of great interest to computer science (e.g., perceptual and cognitive processes in man, and cellular processes that are controlled by genetic information). An understanding of these processes is intrinsically important, and it promises to enrich the pool of basic concepts and schemes that are available to computer science. In turn, application of the current approaches and techniques of the discipline to cognitive psychology and to biosciences promises to result in important insights into natural information processes. To date, most of the work on these processes has proceeded either by modeling them in digital computers and studying these models experimentally, or by using existing theoretical models in computer science (e.g., in automata theory) for the analysis of certain properties of these processes. There is still little contribution from the study of natural information systems to the design and use of computing machines, or to the development of theoretical concepts in computer science.

Scope and Nature of Activities in Computer Science.

The subject matter of computer science can be broadly divided into two parts. The first part covers information processing tasks, procedures for handling them, and a variety of related representations. The second part is mainly concerned with a variety of structures, mechanisms, and schemes for processing information. From the point of view of the practitioner in the computer field, the first part corresponds to computer applications, and the second corresponds to computer systems. There are significant connections between the two parts. Indeed, it is a major goal of computer science to elucidate the relationships between application areas and computer systems.

Computer applications can be broadly subdivided into *numerical* applications and *nonnumerical* applications. Work in numerical applications is mainly oriented toward problems and procedures where numerical data are dominant, such as problems in the areas of numerical analysis, optimization, and simulation. These areas are important branches of computer science. Work in nonnumerical applications is primarily concerned with processes involving nonnumerical data such as representations of problems, programs, symbolic expressions, language, relational structures, and graphic objects. Branches of computer science with major activities in nonnumerical applications are artificial intelligence, information storage and retrieval, combinatorial processes, language processing, symbol manipulation, and graphics.

Computer systems can be partitioned into *software* systems and *hardware* systems. The emphasis of work in software systems is on machine-level representations of programs and associated data, on schemes for controlling program execution, and on programs for handling computer languages and for managing computer operations. Branches of computer science with major concern in software systems are programming languages and processors, operating systems, and utility programs and programming techniques. The emerging branch of computer architecture is concerned with software systems as well as with hardware systems. Other major branches of computer science with a main focus on hardware systems are machine organization and logical design.

Generally, applications-oriented activities in computer science are also concerned with related systems problems; e.g., with higher-level languages and with their computer implementation. Similarly, systems-oriented activities are also concerned with the task environments (e.g., classes of applications

and modes of man-machine interaction) in which the systems operate.

We can identify two major types of activities in computer science:

1. Building conceptual frameworks for understanding the available empirical material in the discipline, via an active search for unifying principles, general methods, and theories.
2. Exploring new computer systems and applications in the light of new concepts and theories.

The first type of activity is analytic in nature; the second is oriented toward synthesis, experimentation, and probing for new empirical knowledge. A continuous interaction between these activities is essential for a vigorous rate of progress in the discipline. The situation is analogous to the interaction between theoretical and experimental work in any rapidly developing natural science.

At present, the theoretical underpinnings of computer science are at an early stage of development. In some areas, theoretical work is mainly oriented toward bringing elementary order into a rapidly accumulating mass of experience, via the introduction of broad conceptual frameworks and analytic methodologies. In a few areas, theoretical work is concentrating on comprehensive analysis of specific classes of phenomena for which formal models exist. Branches of computer science involved in this type of work are theory of computation, automata theory, theory of formal languages, and switching theory. In general, theoretical work in computer science has been diffused over a large number of fairly narrow phenomena. Much of this work has not yet had an appreciable impact on the complex problems of systems and applications that are encountered in the computer field. There is a growing concern, however, with the development of unifying principles and models that are appropriate for understanding and guiding the major constructive and experimental activities in the field. The emerging work in the new area of analysis of algorithms (which includes important approaches to the study of computational complexity) promises to contribute significant theoretical insights into problems that are in the mainstream of the computer field. As computer science continues to grow, theoretical work in the discipline is also likely to grow, not only in relative volume to the other activities in the discipline, but also in relevance to the significant problems in the domain of computer science.

Experimental work in computer science requires extensive use of computers, and it often stimulates new developments in computer design and utilization.

COMPUTER SCIENCE

tion. Typical experimental activities may involve the development and evaluation of a new computer language or the testing of a procedure for a new class of problem. Theoretical work in the discipline relies on several branches of mathematics and logic. A typical theoretical problem may focus on the characterization of a class of computer procedures (e.g., procedures for sorting data), the analysis of their structure, and the establishment of bounds on the storage space and time that they require for execution. The objects of study in this example are computer procedures and their properties. The theoretical treatment of these objects is conducted within mathematical systems that provide the analytical framework needed to obtain the desired insights and specific results. Just as mathematics is used in chemistry (say, to develop theories of certain chemical processes), mathematics and logic are used in computer science to study information processes.

Relationships Between Computer Science and Other Disciplines. The bond between computer science and mathematics is stronger than the normal bond between mathematics and the theoretical component of a science. Computer science and mathematics have a common concern with formalism, symbolic structures, and their properties. Both put emphasis on general methods and problem-solving tools that can be used in a great variety of situations. There are subjects, such as numerical analysis, that are being studied in both disciplines. These are some of the reasons why computer science is widely considered a *mathematical science*.

Computer science is also considered an *engineering science*. The structure of a computer system consists of physical components (the hardware) in the form of electronic or electromechanical building blocks for switching, storage and communication of information, and programs (the software) for managing the operation of the hardware. In the logical design, and the system design of a computer system, the designer is concerned with the choice of hardware and software building blocks, and with their local and global organization in the light of given operational goals for the overall system. These design activities have strong points of contact with work in electrical engineering and in the emerging field of software engineering. They are also important subjects of study in computer science.

Every transition from the specification of an information processing task to a system for implementing the task involves a design process. In many cases, these processes are highly complex, and their effectiveness is strongly dependent on the availa-

bility of appropriate methodologies and techniques that may be used to guide and support them. This is one of the reasons why computer science is concerned with methodologies of systems analysis and synthesis and with general tools for design. This concern is shared not only with engineering, but also with other decision-oriented disciplines such as business administration and institutional planning. There is a more fundamental reason for a close coupling between computer science and a science of design. It comes from the concern of computer science with the information processes of problem solving and goal-directed decision making, which are at the core of design. Processes of this type are objects of study in **artificial intelligence**, a branch of computer science.

Several other disciplines are recognized as having domains of interest which overlap with computer science. One of these is library science. The problems of organizing and managing knowledge, and of designing systems for its storage and retrieval (in the form of documents or facts), are shared between computer science and library science. The activities at the interface between these two disciplines are often identified as part of information science. The main concern of information science is with processes of communication, storage, management, and utilization of information in large data base systems. Thus, the domain of information science is included in the broader domain of computer science.

Another discipline whose domain of interest overlaps with computer science is linguistics, which shares with computer science a concern with language and communication. The study of linguistic processes, and of related phenomena of "understanding," establishes a special bond between computer science and psychology. Psychological research in information processing models of cognition, perception, and other mental functions has a substantial overlap with work in computer science.

The study of certain theoretical questions about processes of reasoning by computer (performing deductions, forming hypotheses, using knowledge effectively in problem-solving processes) is beginning to create points of contact between certain parts of philosophy (logic, epistemology, methodology) and computer science.

The development of computer science has been strongly stimulated by demands for the application of computers in a wide variety of new areas. The challenges created by new computer applications, and the constructive attempts to meet them, are important factors in the growth of computer science. The exploratory activity in the discipline, as it

interacts with other disciplines in the development of computer applications, results both in a better understanding of the power and limitations of current knowledge in the computer field, and in the identification of new problems of information processing that require further study. At a more practical level, the exploratory work on computer applications is contributing to the solution of significant problems in various disciplines that could not be approached without the introduction of computer methods.

There is a large "surface of contact" between computer science and the disciplines where new computer applications are being developed. Virtually all disciplines are involved in this contact. The nature of the contact is similar to the relationship between mathematics and the physical sciences; this relationship involves the representation of scientific problems in mathematical systems wherein the problems can be studied and solved. In the case of computer science, the contact involves the representation of knowledge and problems of a discipline in forms that are acceptable to computers, and the development of computer methods for the effective handling of these problems. Since computers can be made to represent and manipulate problems of enormous variety and complexity, it is likely that the extent of fruitful contact between computer science and other disciplines will be much larger than the contact between mathematics and the "mathematics utilizing" disciplines. In particular, it is likely that the role played by computer science in behavioral and social sciences, the professions, and the humanities will be similar to that played by mathematics in the growth of the physical sciences.

An important application for computers, which is of special interest to computer science, is in the design of more powerful, efficient, and easy-to-use computer systems. The use of computers in the study of computers and in their improvement is a powerful means for gaining the knowledge and insights that computer science seeks, while at the same time the field is being bootstrapped.

From the previous discussion it can be seen that computer science has two types of interface with other disciplines: The first type is characterized by a *shared concern* with subjects of study that are of intrinsic interest to computer science. Here there is an area of overlap between work in computer science and work in other disciplines. Mathematics and electrical engineering have this type of interface with computer science. To a lesser extent, such an interface exists between computer science and the decision-oriented disciplines (e.g., business administration; institutional planning), library science, linguistics,

psychology, and philosophy. The second type of interface includes disciplines in which new computer applications are being explored. The main role of computer science in these activities is to support and enhance work in a discipline. Practically all disciplines that involve some kind of intellectual activity have this type of interface with computer science.

The Internal Structure of Computer Science. The pattern of relationships between computer science and other disciplines is likely to change as the internal structure of activities in computer science continues to change. While the overall structure of the discipline is beginning to attain considerable stability, its detailed internal structure is less stable, and the relative emphasis that various subdisciplines are receiving is still far from stabilized.

The conception, formulation, computer implementation, analysis, and evaluation of procedures (algorithms) for a broad variety of problems constitute a major part of the activities in computer science. Closely associated with these activities are efforts to develop schemes, means, and tools for building and executing procedures—such as languages, major principles for structuring procedures, programming mechanisms, computer organizations, and design aids to facilitate these efforts. In addition, a significant amount of effort is directed to the design of advanced systems—software and hardware. All these activities have important connections with several theoretical efforts in the field, some in application areas and others in the analysis of algorithms, in formal languages, automata theory, switching theory, and systems analysis.

An outline of the major areas of study in computer science (and some of the major relationships among them) is presented next.

1. *Representations in Computer Language Of Problems, Data, and Procedures in Various Application Areas.* The main problems in this area are to find solution methods for classes of problems in different domains of application, and to formulate them in a suitable computer language. As mentioned previously, the two major families of applications in the discipline are numerical and nonnumerical applications.

2. *Theory of Computation and Analysis of Algorithms.* Work in this area is concerned with computability, recursive functions and properties of classes of procedures (algorithms) such as complexity, validity, and equivalence. It is related to work in (1).

3. *Higher-Level Languages for Various Applica-*

COMPUTER SCIENCE EDUCATION

tion Areas, Schemes for Structuring Data and Procedures, Language Descriptions, and Translation Schemes. Work in this area is central to the facilitation of man-machine communication and it has a strong impact on computer applications. It is related to work in computer design and also to work in (1).

4. *Machine-level Languages, Storage Schemes, and Programming Mechanisms.* This area is concerned with the art of programming computer hardware. It interfaces with (3) and to a lesser extent with (1), and also with (5).

5. *System Organization Schemes, Executive and Control Mechanisms, and Computer Design Processes.* Theoretical activities related to this area are system analysis and simulation (at the hardware/software configuration level), automata and switching theory (at the logical design level), and theory of digital circuits and devices (at the machine component level). This area is strongly related to professional activities in computer system design.

6. *Theory of Formal Languages, Automata Theory, and Switching Theory.* Theoretical activities in these areas are concerned with properties of computer languages, computer mechanisms and their realizations. They are related to work in (2), (3), and (5).

Computer science is a young and rapidly expanding discipline. In a period of less than 15 years, it has succeeded in establishing its distinct identity in universities and in laboratories throughout the world. One of its recognized roles is to provide the intellectual guidance needed for the understanding and development of the computer field. Another role, which is likely to grow in significance in the coming years, is to contribute to an understanding of the impact of computers on other disciplines and on society in general.

REFERENCES

- 1968. National Academy of Sciences. "The Mathematical Sciences: A Report," Publication 168 1, Washington, D.C.
- 1969. Hamming, R. W. "One Man's View of Computer Science," 1968 ACM Turing Lecture, *Journal of the ACM*, Vol. 16, No. 1 (January), p. 5.
- 1970. Wegner, P. "Three Computer Cultures-Computer Technology, Computer Mathematics, and Computer Science," in Walter Freiberger (Ed.), *Advances in Computers*, Vol. 10. New York: Academic Press.
- 1971. Amarel, S. "Computer Science; A Conceptual

Framework for Curriculum Planning," *Communications of the ACM*, Vol. 14, No. 6 (June).

S. AMAREL

COMPUTER SCIENCE EDUCATION. See **EDUCATION IN COMPUTING SCIENCE**.

COMPUTER SECURITY. See **CRIME AND COMPUTER SECURITY; DATA SECURITY; and SECURITY OF COMPUTER INSTALLATIONS, PHYSICAL**.

COMPUTER SYSTEMS

For articles on related subjects see **ARITHMETIC-LOGIC UNIT; CENTRAL PROCESSING UNIT; CHANNEL; COMMUNICATIONS AND COMPUTERS; COMPUTING CENTER; COMPUTER NETWORKS; INFORMATION SYSTEMS; INPUT-OUTPUT DEVICES; INTERRUPT; MEMORY: Main; MEMORY: Auxiliary; OPERATING SYSTEMS; PROCESSING MODES; SOFTWARE; and STORAGE HIERARCHY**.

For articles on related terms see **APPLICATIONS PROGRAMMING; COMPILER; DATA BASE AND DATA BASE MANAGEMENT; DATA STRUCTURES; EMULATION; LOADER; MICROPROGRAMMING; MULTIPROGRAMMING; OBJECT PROGRAM; SOURCE PROGRAM; SYSTEMS PROGRAMMING; TIME SHARING; and UTILITY PROGRAM**.

A modern computer system is one of the most complex and wonderful achievements of mankind. Its complexity is indicated by the fact that the equipment of a single computer may easily contain over a million identifiable parts, all working together at very high speeds and with remarkable reliability.

As with any complex configuration, it is helpful to consider a computer system as composed of subsystems, each made up of various major components. Because it is more easily visualized, we will begin with a description of the equipment or *hardware* subsystem and then proceed to consider the programming, or *software*, subsystem. Finally, we

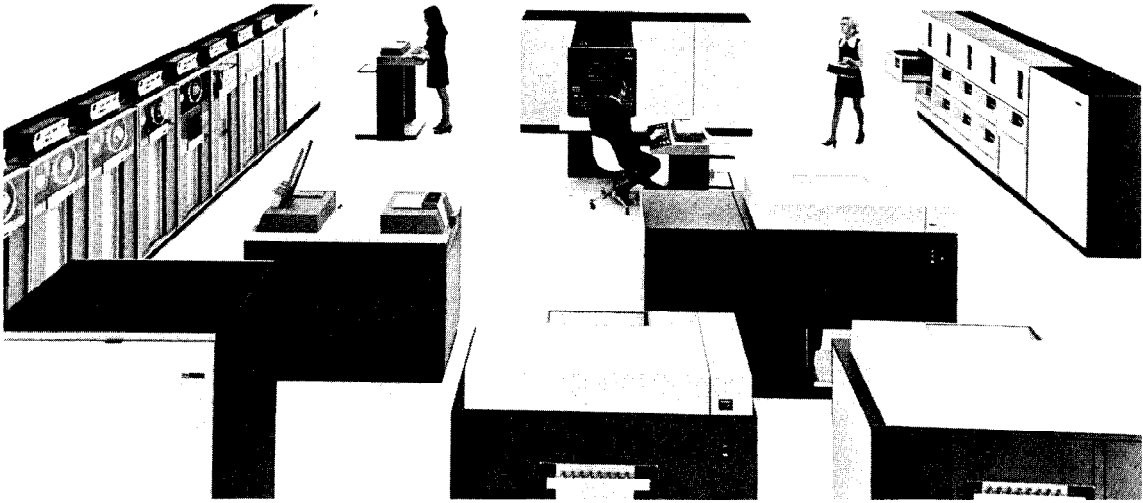


Fig. 1. An IBM 3701155 computer system.

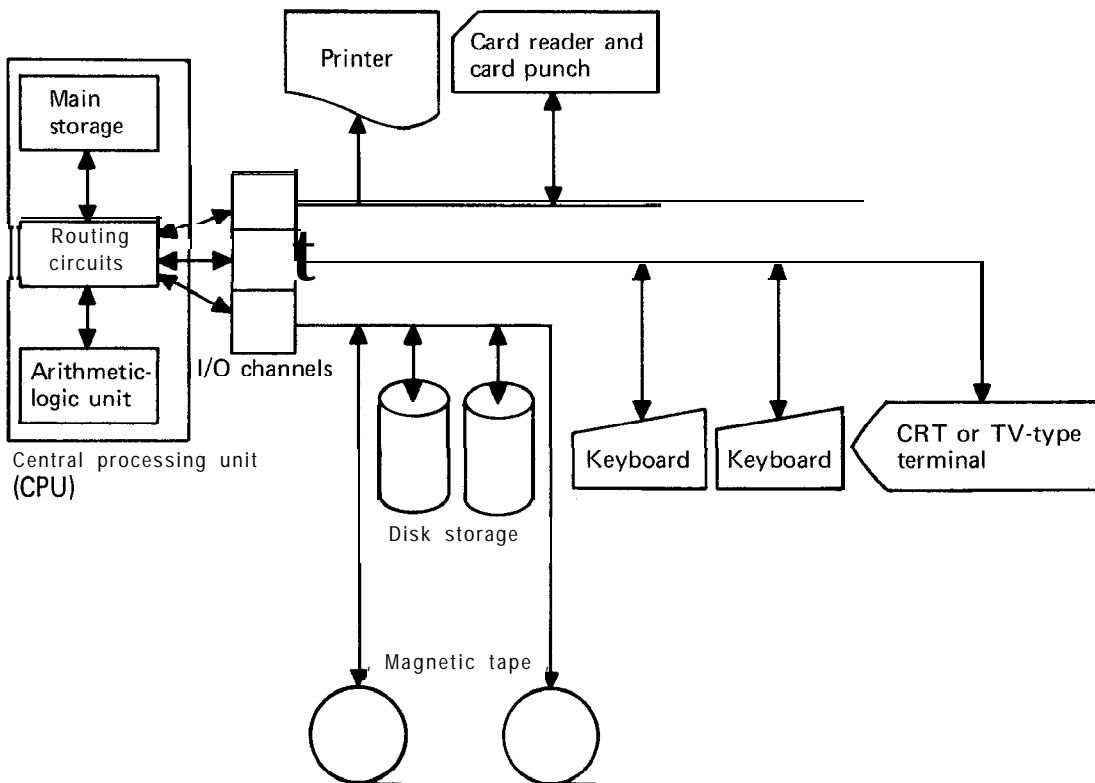


Fig. 2. General organization of the hardware subsystem of a typical digital computer.

COMPUTER SYSTEMS

will interpret how these appear first to computer users and then describe the programs created by the users.

The Hardware Subsystem. Fig. 2 shows the major hardware components. These can be classified into a number of categories:

Transducers. These are hardware devices that change information from one physical form to another and hence serve as communication links between the computer and its environment. Examples of transducers are card readers, graphics terminals, high-speed printers, typewriter terminals, and plotters, all of which transform human-readable information into an electrical form suitable for computer processing (or vice versa).

Storage Devices. These devices in a computer system store not only its data but its instructions ('programs) as well. Storage devices come in many sizes, speeds, and costs. They range from the extremely cheap and slow (e.g., punched cards) to devices whose speed make them suitable as on-line (i.e., directly connected to the computer) auxiliary storage devices (e.g., magnetic tapes, disks, and drums). There are also devices fast enough for use as primary storage (e.g., magnetic cores and electronic circuit storage).

Transformation Devices. These are the circuits that do most of the "work." They are typically concentrated in a structure called an "arithmetic-logic unit" (ALU), which contains an adder circuit augmented by shift and other control features that together implement almost all of the system's arithmetic and processing operations. The ALU also contains the circuitry for program control which directs the machine from one of its instructions to the next with provisions for testing various conditions and *branching* (i.e., causing a change in the program sequence from strict progression in the written program). All these ALU functions require the use of extremely fast (but expensive, and hence small) storage, the ALU registers. The ALU uses the registers as a sort of scratch pad to jot down results that will be transferred later to primary storage.

Communication and Control Devices. Communications circuits include the networks or busses that direct the flow of information between the other functional parts of the hardware subsystem. For instance, the input/output (I/O) channels control the flow of information between the transducers, auxiliary storage devices, and main storage. Other routing circuits control communication between main storage and the ALU. The control circuitry generates timing signals in various complex arrange-

ments that specify at what times information is moved from place to place in the system.

Another classification scheme divides the hardware subsystem into internal components and external components. The internal components are the ALU, with its associated registers and routing circuits, and main storage. The "internal computer" is often referred to as the CPU (although sometimes just the ALU registers and routing circuits are regarded as the CPU). All other hardware devices are part of the *peripheral*, or *I/O, subsystem*.

Now that we have introduced the hardware components of a computer system, how do they work together? Let us trace the path of a program through the hardware. The reader is advised to follow the description by tracing the events through the paths and facilities of Fig. 1. The program (say, as a deck of punched cards) must first be physically translated into electric-signal form, which is done by the card reader, a transducer device. To be executed, the program must be in the main storage, since this is usually the only store available to the CPU. Both the main store and CPU are very fast devices. To keep up with their speed, they should be fed program and data from a reasonably fast storage. Since the card reader is slow, its information is not moved directly to main storage for processing. Instead, it is first moved by an I/O channel to the intermediate-speed disk (briefly passing through main storage in the process), where it is held until the CPU is ready to work on it. At that time the program and its data is moved via an I/O channel from the disk store to main storage.

Once in main storage, the program is accessible to the CPU and can be executed. During execution, most of the storage accessing is to the main store. However, the program is capable of receiving/sending larger volumes of data from/to the auxiliary stores (disk or tape) via the I/O channels. When the program finishes executing, its results are moved (again by the channels) back to auxiliary storage, and finally they leave the computer system via a transducer such as a high-speed printer.

Because of the slow speed of I/O operations relative to central processing, it is best for efficiency if they can proceed concurrently (usually on different jobs) rather than in strict time sequence. A typical system has only one CPU, and its attention is required to service I/O operations rather frequently but briefly each time and at unpredictable times. The sharing of the single CPU between I/O and central processing is made possible by an *interrupt* scheme that permits channels to suspend ongoing CPU

operations, give the required brief service to I/O or other external requests, and then return to what it was doing.

This completes the general explanation of how the hardware system components interact, but a few fundamental questions still remain. How does the computer system “know” when to transfer information, what information is required, where should it be stored? As we will see, this guidance is supplied by the second subsystem, the software.

The Software Subsystem. Unlike the hardware components, we can’t point to a specific physical object and say, “this is a part of the software subsystem.” The software simply isn’t composed of physical devices; it is composed of *programs* and certain *data structures*. The programs include those that the computer users write, which are generically called **application programs**. Another category of programs, of primary interest to us now, is called **system programs**, the purpose of which is to give all programmers convenient ways to manage and control the hardware, software, and stored data.

During the time that a user’s program is being processed, it makes requests for stored data, executes instructions that process the data, and generally controls at least a portion of the computer system’s hardware resources. Hence, while a user’s program is being executed, it is a part of the active software subsystem.

The next portion of the software that we consider is the **operating system**. Unlike applications programs, it is a permanent part of the computer system. Its function is to control the execution of the other resources in the system. The operating system is a collection of interrelated system programs. These can be classified into three groups.

Control Programs. Typical examples are: a reader-interpreter program that reads input from a transducer, translates certain control and scheduling information, and stores the program on auxiliary storage (disk); and a scheduler program that determines which job the computer system should service next. Once the scheduler has selected the *next* job, its program is placed under the control of an initiator/terminator program, which obtains the resources (such as main storage) necessary for the execution of the program, starts it up, and “cleans up” after it has been completed. During execution, the job delivers its output to a disk. Later, a system “writer” program moves the output to a printer for delivery in human-readable form.

Installation (or manufacturer) Supplied Programs for User Convenience. These programs fall into var-

ious categories: first there are **translators**, like *compilers*, which are programs that translate user-written programs from the **source language used by the** programmer into a language the machine can execute. Second are **loader** programs that place the translated programs into main storage in a form that the computer system can execute. Third are **utilities**, which are programs that perform frequently required tasks such as sorting and merging two or more lists, moving large masses of data from one place to another within the system, etc.

System Data-Management Programs. These programs keep track of what is in the system and where it is located, and use various means to store and access the data efficiently. For instance, when a user’s program calls for data, the data-management programs locate and fetch the data to the requesting program. For every data collection (file or data set), data-management programs record who is permitted to use the data, who is currently using it, what is being done with it, whether or not the data should be retained in the system after the job ends, etc.

The third part of the software subsystem comprises the **installation libraries**. These contain data and programs that are useful to a wide spectrum of users. What is specifically contained in the libraries will vary from installation to installation; a manufacturing company’s library would probably contain an up-to-date inventory of its products, a list of recent orders, etc. An airline reservation system’s library would probably contain a schedule of all flights with arrival and departure times, destinations, flight numbers, number of vacant seats on each flight, etc.

System software is designed in many sizes and complexities. A modern computer system that allows **multiprogramming** (more than a single program executing concurrently) or **time sharing** (several users interacting with the system at typewriter or TV-type terminals at human reaction times) would have a software subsystem that contained all the features previously discussed, and probably some others. However, in a less sophisticated computer system, a good deal of what we have described as software functions is done by human intervention (either by the operator or by the user himself).

We have described a computer system as a collection of two interrelating subsystems, hardware and software. There are, however, certain aspects of the system that do not fall into either subsystem; an example is a **microprogram**, which has been termed “firmware.” Microprogramming, as the name

COMPUTER SYSTEMS

implies, is a type of programming. Microprograms directly control the sequencing of computer circuits at the detailed level of the single instruction. Organizing the control hardware in a microprogrammed structure rather than as wired circuitry has several advantages. First is economy of circuitry if the machine must have complex instructions. A second advantage is that it is possible, by microprogramming, to produce an emulator, i.e., a set of microprograms that makes a given machine have the same appearance to a programmer as some other machine! This permits the same machine to run programs written for either itself or the machine it is emulating at reasonable efficiency. Yet another advantage is to produce faster operations for the special functions that are microprogrammed rather than programmed in the usual manner. There are, however, some negative aspects of microprogramming, such as the highly specialized knowledge needed and the great tedium of writing microprograms. Also, although microprograms are faster than doing the same functions with software, they are slower than using wired control circuitry.

The User's View of the Computer System. Let us now imagine that we are the users of a large, modern computer system. How does this system appear to us? Our first problem is gaining access to the system. First, we must arrange with the computer personnel to issue an *account number* to us. This will be used in a number of ways; for example, to keep track of our use of computer time and resources so that we (or our employer) may be charged our fair share of the system's cost. When we have an account number, we can attempt to interact with the computer system. To do this, we must write a program in one of the many languages available in our particular system. Let us assume that we have written a **Fortran** program, punched it on a deck of cards, appended a deck of data cards that the program is to process, and are ready to submit the entire deck to the computer system. We then take our deck and surround it with some "job control cards." These tell the operating system our account number, the language our program is written in, and other information the system requires. We put a rubber band around the entire deck, which is now called a "job," take it to the computer center, and submit it to a computer operator. We then wait until the computer operator returns the deck and its associated printed output.

But wait, you say, what happened to the CPU, main storage, I/O devices, etc., that you were talking

about before? Amazingly, almost the entire computer system, with all its functional characteristics, can be ignored by the average computer user (however, all this will reappear when we see how the computer system appears to our **Fortran** program). To the average computer user, the entire physical computer system may be regarded as a "black box" into which he submits his input (program) and from which he receives his results (output).

The user, however, does perceive something extremely important about a computer system. The entire computer system (hardware and software) is itself but a subsystem of a much larger system that is the environment in which the user works. This includes the computer operator, the policies of the computer center in scheduling and billing of jobs, etc.

Program View of a Computer System. Once the user leaves his program with the computer operator, how does the program interact with the system? First the program is placed in a card reader (a transducer), the card reader moves the information punched on the card deck to an auxiliary storage device (such as a disk) where it is stored by the reader-interpreter system program. The reader-interpreter is concerned primarily with the information contained on the job-control cards. Accounting information is placed in a file that will later be used to compute the individual user's bill. More importantly, from the point of view of the computer system, information describing various scheduling parameters is placed in a set of tables. These tables are scanned by the scheduler, which is another system program, to determine which job should be executed next. The tables contain all the information the scheduler needs to construct a relative priority ordering of all the programs waiting for service. When the scheduler determines that it is time for the user's program to be executed, the following events take place:

1. The control of the program is passed to another operating system routine, the initiator-terminator. This program scans the job-control card information supplied, determines from it that the program is written in **Fortran**, and hence must be translated into a language the computer can execute (machine language). The initiator-terminator calls for the **Fortran** compiler (the program that performs the translation) and then starts it executing. The **Fortran** compiler, using the program as input data, produces a machine-language translation. In the

COMPUTER USER GROUPS

course of the translation process, the compiler will call on various other programs in the operating system for help in doing tasks such as allocating temporary auxiliary storage space. When the Fortran compiler finishes the translation, it stores the resulting machine-language program (often called an "object module") on auxiliary storage, and then notifies the initiator-terminator that it is finished. The initiator-terminator releases the space the Fortran compiler occupied in main storage, and then calls another operating system program, the loader.

2. The loader does some necessary processing on the object module and brings it into main storage so its execution can begin. As the program executes, it will interact with various operating system programs that fetch the data it requires, supply the program with any auxiliary storage it requires, etc. When the program finishes executing, it signals the initiator-terminator. The program's output is stored on auxiliary storage. The initiator-terminator "cleans up" after the program and supplies the operating system's "writer" program with the program's output. The writer moves the output information to a high-speed printer. The computer operator then "bursts" (separates) the output and returns it with the punched-card job deck.

Summary. To summarize, a computer system is best considered as a collection of resources consisting of two broad classes, hardware and software. Work for the system consists of *jobs*, which are programs and their data prepared by programmers. Both hardware and software are managed by a carefully designed collection of system programs called an "operating" system, which controls the flow of jobs through the system, furnishes services such as language translation, and provides various utilities.

A computer system, through its large capacity storage, can serve as a repository for procedures (programs) that can be shared productively by members of an industrial or educational community.

REFERENCES

- 1971. Bell, C. G., and A. Newell. *Computer Structures, Readings and Examples*. New York: McGraw-Hill.
- 1973. Hellerman, H. *Digital Computer System Principles*, 2d ed. New York: McGraw-Hill.

H. HELLERMAN AND I. A. SMITH

For article on related subject see MANUFACTURERS, COMPUTER.

For articles on related terms see CUBE; GUIDE; JOINT USERS GROUP (JUG); SHARE; and UNIVAC SCIENTIFIC EXCHANGE (USE).

This brief history of the rise, maturation, and old age of computer user groups represents a sociological textbook example of any volunteer organizational entity. After arising from a need and developing into a forceful activist group, a gradual decline has ensued as administrative paralysis set in. Computer user groups began because the manufacturers only barely understood how to support what they produced; by the mid-1960s the need was greatly reduced, and at the present time there is almost no need at all.

History. The precise origin in 1955 of the first user group, SHARE, is obscure. Prior to 1955, users of the IBM 701 in the Los Angeles area had worked cooperatively on PACT-I, a primitive automatic programming system. While working on PACT-IA for the forthcoming IBM 704, the users felt an urgent need to create a united front against a proposed IBM assembler, since it was far short of being as useful as it should have been. A meeting was hastily called, and the first formal user group meeting was held in a basement room at the RAND Corporation's headquarters in Santa Monica, California, during the week of August 22, 1955 (see Armer, 1956).

Installations represented were a fitting cross section of the large-scale, scientifically oriented computer community of that era. There was one government agency, NSA (National Security Agency), three government-sponsored research establishments (RAND, Los Alamos, and Livermore), eight aerospace organizations (Boeing, Curtis-Wright, Hughes, North American, United Aircraft and three Lockheed divisions), three industrial giants (General Electric, General Motors, and Standard Oil of California), and IBM (Steel, 1956).

Just a few months after the founding of SHARE, a group of IBM users of commercial computers, recognizing the idea of a user group, banded together to found GUIDE. Today this organization, whose membership requirements roughly parallel those of SHARE, far exceeds all others in head count, activity, and energy. Other early SHARE "spin-offs" included users of Control

COMPUTER USER GROUPS

Data equipment, the GE 600 line, and Philco Transac equipment. Of these user groups, only VIM (the CDC 6000 series) survives. From this beginning, a regular alphabet army of similar organizations has emerged; for example, COMMON, DECUS, HUG, USE, CUBE, SEAS, TAG, ECHO, DUA, CSSHARE, and the IVY LEAGUE. Each merger reduces the count; each new industry entrant increases the count. There is even a group of groups, JUG (Joint Users Group), although not all user groups are represented in it. A number of large groups are specifically unrepresented because of attempts by manufacturer employees to pretend to represent users.

Purposes. In an era before software was sold, a fundamental purpose of a user group was the swapping of home-grown software. Before manufacturers supplied subroutines, users had little but their own ingenuity on which to rely for the countless routines necessary to keep a system running. Such routines as a memory dump from Phillips Petroleum, an internal Sort from UCLA, an assembly program from United Aircraft, Bessel function subroutines from General Electric, and a CRT package from General Motors, all crossed and recrossed the country, spread by word of mouth and the SHARE library, founded and operated by Ben Faden of North American Rockwell Corporation.

It was not beyond the pale for a user group to generate the specifications and do most of the implementation for an entire operating system. One such example was SOS, the SHARE operating system, which was implemented for the IBM 709 and later for the IBM 7090. But the increasing complexity of today's systems has made it virtually impossible for a loosely organized, volunteer association to successfully implement large projects. To survive, user-group purposes had to be altered. The current SHARE purpose is stated in the group's by-laws as "... to foster the development, free exchange and public dissemination of research data pertaining to SHARE computers ... in the best scientific tradition." It implies that the group now exists to generate a climate for the exchange of data rather than for the original creation of new data.

Despite this disclaimer of innovative objectives, the general view is that the user groups have become little more than underpowered lobbying forces, attempting with only marginal success to translate user needs into product specifications. Instead of the aggressive developmental attitudes of the late 1950s and early 1960s, the groups now display reactive and defensive tendencies.

Membership. Membership in user groups is generally confined to those installations that have installed or have on order the specific hardware, program, or service which the group is organized to market. However, groups such as CDC's VIM relax this requirement of eligibility to permit at least attendance at meetings by all who express interest in the "system." Although the relaxation of the rule is attractive, since it invites extended participation, this broader membership base may lead to more emphasis on sales prospects than on the interests of real customers for those groups under tight control by their vendor. This, then, is a sales device, a perversion of the reasons why users organize.

Membership counts vary widely, from as few as 50 or less to as many as the 1, 100 or more installations now members of GUIDE, the IBM commercial users' group. Retaining membership requires little more than a declaration of interest, although a few of the more formal groups require representation at one meeting every year or two.

One still unsolved problem is that of the bona fide nature of an application for membership. With the industry's reluctance to release sales data, a user group has almost no way to verify that the statements on the application are genuine. It has not been unknown for a paper company, with no resources, to join a user group before its corporate certificate of incorporation was placed on file.

Legal Status. An often-used greeting at user-group meetings is: "Fellow Conspirators!" The legal status of user groups is vague; while no group intentionally frames a conspiracy to control the market, from time to time some of the groups have been on thin ice. The exact status of user-groups is questionable and will doubtless remain undefined, since nobody really is very interested in testing the matter in court.

A few user groups have incorporated to obtain the protections of corporate law for their officers. While accusations of secret societies and cabals have been made, no outsider has yet taken the matter seriously enough to use the courts to obtain entry, although in one case it was actually contemplated. From a tax viewpoint, a user group ought to be a not-for-profit, tax-exempt organization of a scientific and/or educational nature; unfortunately, the U.S. Internal Revenue Service (IRS) does not agree with this position. IRS rulings are rarely clear-cut, but the point of contention appears to be the restrictive nature of the membership rules. The IRS emphasized this point in taking action to withdraw the 501(c)(3) tax exemption from one user group, al-

though this has not been generally applied to all user groups.

Practices. A first visit to a user-group meeting is equivalent to an introduction to a three-ring circus—exciting, stimulating, confusing, and almost overwhelming. Activity swirls from early in the morning to late at night; 20 meetings may be running in parallel; social events continue into the “wee” hours; and small knots of uptight people are seen huddling in corridors, engaging in apparently strategic planning. Actually, what is happening consists of small, face-to-face technical confrontations; limited-size working parties planning implementation and specification priorities; medium- to large-sized groups listening to technical presentations, with a minimum of interaction; and formal assemblies that are likely to be hearing sales pitches of the “you’ll love it when you get it” variety, a term originated by Carl Reynolds of IBM at a SHARE meeting.

If a representative is to be more than a listener, he must learn to match his installation’s needs with the information dispensed, which is not always easy because meeting agendas are usually broad in scope. He must seek out the intimate, unlisted (but critical) meetings that are held at odd hours, and to do this he must have experience. There is a whole class of “nonmeetings,” which offers a convenient way for a vendor to try out ideas, avoid premature disclosures, and sidestep internal politics. An example is the 1970 CUBE meeting in St. Louis, where certain key Burroughs users “did not meet” in a hotel room with Burroughs management and technical people. From that meeting came the B-6700 PL/I compiler effort.

From a user’s viewpoint, the happiest situation is the one in which the technical people meet quietly and engage in dialog with the product development team. More often, however, users find themselves faced with a marketing representative who can speak technical jargon but who exhibits considerable skill at sidestepping issues, avoiding promises, and evading commitments. When users and developers are not subjected to such routines, the relationship is mutually satisfactory. However, manufacturers generally try to avoid this situation. Vendors have nightmares about permitting development teams to make implementation decisions based on technical issues. Current development costs are so high that even the smallest implementation decision may require lengthy examination from the marketing viewpoint. What needs to be answered is always the same question: “If this is implemented, do we either avoid

the loss of some account or gain the sale of additional hardware?”

Accomplishments. What is actually accomplished by user groups? The record is erratic, and it appears that the group effectiveness curve is dipping sharply. What was once a viable entity that created new compiler languages, operating systems, and applications packages has today become a patch-and-fix and complaint exchange, with little creative activity. The vastness of today’s systems, the size of the vendors, the difficulties of sustaining voluntary action against full-time workers, and the rising expenses involved have all combined to squeeze the user group’s effectiveness.

As a result, a handful of dedicated people working part time are gradually being subordinated to paid professionals. The user has almost no opportunity today to alter significantly the primary thrust of product developmental efforts; those lines are set by marketing requirements, competitive timings, and product life cycles. All the user group can do is perform minor cosmetic surgery on the specifications, detect and note the gross functional errors, and flag the basic implementation faults when the product is released to the field.

Does the user group have any lasting effect? Is there a positive return on the investment of time and money by the user community? Most outside observers doubt it. One critic has stated that all user groups ought to be dissolved six months after the first machine of its series has been installed; at that time its problems will either have been fixed or will never be fixed, no matter how long the system is out. Although the future seems technically bleak, there is no reason to conclude that these unique groups will quietly fade away. The facade of effectiveness, the pretense of working together, the sales value to the manufacturers, and the social amenities make it far too pleasant an experience for any abrupt termination. As with so many formal institutions whose time has passed, the body continues to expand and look alive, even though the soul and spirit have long since departed.

REFERENCES

- 1956. Armer, Paul. “SHARE-An Eulogy to Co-operative Effort,” RAND Report P-969 (October).
- 1956. Steel, T. B. *SHARE Reference Manual*, p. 0.1-01.

P. DORN

COMPUTER UTILITY

For articles on related subjects see ARPA NETWORK; COMMUNICATIONS AND COMPUTERS; COMPUTER NETWORKS; COMPUTER SYSTEMS; DATA BASE AND DATA BASE MANAGEMENT; and TIME SHARING.

The expression "computer utility" has come into use by analogy with other public utilities such as those that supply water and electricity. These **utilities** provide, often for metered payment, a public service almost everywhere. Electricity is delivered to one's home and one may use it for general purposes, provided the bill is paid. The analogy is made between electric power and computing power; the intent is to make computing power or capacity available to all comers at their convenience and for their purposes, provided they pay for it. The usual means envisaged for providing this service is the use of terminals such as teletypewriters connected to a computer by telephone lines.

Early experiments that led to the idea of the computer utility emphasized the provision of scientific calculating power for people who were (more or less) skilled at computer use. The great value of the prepackaged program for the untrained customer was in a sense an accidental discovery, and it is this which leads to the vision of the computer utility as a provider of all kinds of services to all kinds of people. Household bookkeeping, personal records of all kinds, public inquiry services (even access to an encyclopedia such as this via a computer rather than a book on a shelf) are all among the facilities that have been suggested for computer utilities.

The analogy with other utility services must not be pushed too far. For physical reasons, many public utilities are provided on a local monopoly basis; insofar as a computer service can be considered a utility at all, its classification as such depends on its coordination with telecommunication facilities, which are themselves usually provided as a public utility. (For this reason, however, government regulation of computer utilities is an important subject of discussion, particularly in the United States.) Furthermore, for most public utilities, there is little difficulty in insuring that one customer's activities will not interfere with another customer's getting what he pays for. The electricity user, for example, needs no personal equipment located in the power station, and the power station contains no equipment dedicated solely to his personal use. Moreover, the power channel from the user's equip-

ment to the power station is very narrow and may readily be controlled (e.g., by a fuse). In the provision of computing power, by contrast, the computer is from time to time recognizably doing a particular customer's work, and there is a possibility of interference between one customer's work and another's. This interference may be caused either from sheer overloading of the machine (taking so much of its capacity that too little is left to give good service to others) or more indirectly as a result of accident or sabotage by altering another customer's data or programs.

These points lead to a number of requirements for a computer utility, each of which will be introduced briefly and then discussed in more detail. First is the requirement for very adequate and reliable *protection*. It is necessary that a computation done on behalf of one user will in no way alter another's material or have access to it illegitimately. Neither can one user's computation be allowed to affect noticeably the performance of the system as a whole, i.e., the rate at which it does work for others.

The second requirement is *reliability*. A computer utility as ordinarily conceived must be able to store a user's information and to give it up on request, as well as just being able to do computations. The system will not be used unless customers can trust it to retain information reliably and permanently, even if there are occasional failures of equipment. A customer will not pay to have his computations wrongly executed or his information mislaid. He is not interested in the mechanisms for insuring this, but he is interested in their effectiveness.

The preceding two requirements may be regarded as basic. No computer utility will have the confidence of its users unless they are satisfied that these requirements have been adequately met. Equally, no computer utility will make money for its proprietor unless customers are prepared to use the system in sufficiently large volume. To insure that they will do so, it is necessary to meet several other requirements, which, although the economic motivation for meeting them may be just as strong, should be recognized as being in a different class.

The first of these is generally termed *programming generality*. This is a name for a means to an end, the end being that it should be easy to make successive or joint use of possibly a considerable number of pieces of program without getting into enormous difficulty over the process of connecting them together. At one level, it should be possible to put together without difficulty a package for maintaining a data base about sewerage connections in a

own, together with a package for drawing maps. At another level, it should be possible to plug together a subroutine for working out square roots with any mathematical program.

The second requirement in this class is for *predictability of performance*. A computer utility will be unattractive to its customers if the cost of a particular use is unclear, even if a similar use was made yesterday, or if elapsed time needed to perform the work is not definite. The utility must be predictable in both performance aspects.

A third requirement, which is worth mentioning at this point, although it is a requirement that bears more on the provider of the system than on the user, is that computing power of the system be readily enhanceable. It must be possible for the vendor of computing power via a computer utility to provide additional computing power when it is needed, without undue disturbance to his existing customers. He must be able to enhance the equipment used without either shutting the service down for a while or changing the way in which the computer has to be used.

All user and provider requirements present considerable challenge to a computer utility, and the remainder of this article will say a few words about each in turn.

Protection. If in a computer utility it were required to provide only brute computing power for service of a user, it would be sufficient to provide protection that insured that the work done for a particular user could not directly interfere with work being done for any other user or with the mechanisms that provide the entire service. However, it is usual to think of the user of a computer utility having available to him, possibly for a fee, a considerable number of programs. The proprietary nature of these programs must be protected if the owner of them is going to put them out for service.

Accordingly, protection systems must permit users to have access to programs without copying them, and programs must exist that will automatically and safely bill their users on every occasion. It must not be possible for a user to call a proprietary subsystem in such a way that it does the work for him without billing him properly. It must also be possible for the owner of a program or subsystem to stipulate which other customers of the utility may use it. Similar remarks apply to stored data bases.

Thus, it is evident that customer protection requires more than the simple encapsulation of the activities of a particular user while he is engaged in them. The imposition by the owner of a program or

specific data of protection restrictions should be implementable by that owner directly, without his having to request the proprietors of the computer utility to do it for him. If this requirement is not met, the administrative burden is likely to impede effective exploitation of computer service. It would not be appropriate here to discuss at length the detailed techniques for effecting the protection required. The references give some pointers.

Reliability. Reliability of information storage poses very high technical requirements. If a system is to be trusted by users to retain their information indefinitely, it must be capable of providing a much higher degree of integrity than most users would ordinarily consider applying to data themselves in a more direct way. A user of an ordinary computer system will take steps to keep backup copies of his information, in proportion to the value he places upon it and to the difficulty (which only he knows) of recomputing it. He may be prepared to take a risk sometimes, but he will be most displeased if an automatic system assumes that prerogative. Keeping backup information of very diverse sorts for different people on a really large scale is a problem not yet solved. It is made more difficult by a reasonable desire on the part of the utility to provide the degree of safety that a user is prepared to pay for, no more and no less.

It is a question of policy whether the integrity system should protect the user against his own mistakes rather than against errors by the utility itself. This involves the relation between backup storage and performance failure—no matter how caused—and cheap archival storage for deliberate use. Although the two functions are logically quite distinct, the physical media used (usually magnetic tapes) are the same, and the required data organizations are at least similar. It is to some extent an open question how far the two functions can be given a common implementation.

Reliability is an area heavily dependent on the currently available storage technology, in which elaborate systems are very likely to become obsolete as technology progresses. Simple systems are, however, likely to be severely restrictive. For example, a very simple approach is to permit a user to request preservation of his material on magnetic tape centrally, or to permit him to request its preservation locally on a (simpler) tape driven directly from his terminal. However, this works only if there is a solid distinction between the material belonging to the system (automatically preserved) and material belonging to users (their own responsibility). Since one

COMPUTER UTILITY

of the most attractive attributes of computer utilities is the sharing of material, this distinction is not admissible. Reliability could be severely questioned if something went wrong with shared material and the utility had to appeal to the proprietor for a backup copy. Both the owner and the user of shared material must have confidence in the integrity of the central system. Current approaches to these problems depend on automatic means of recovering reserve copies when either the system or the user notices that there is something wrong, so that the worst experience of the user will be a slight delay in his work. Ideally, recovering a data file should be as easy as redialing an abortive phone call.

Programming Generality. Programming generality places requirements on languages and system structure. It is commonplace to find that programs exist for doing the kind of calculation one wants, but that either they will not fit into the rest of one's program structure or will require an **inconvenient** (for the intended purpose) data organization. These problems lead to heavy and unnecessary programming costs. The avoidance or partial **avoidance** of such problems requires that there be **discipline** and convention in the entire structure of system and user programs. It is not clear to what extent this can be reconciled either with efficiency or with the possibility of progress. Programmers will have to be as disciplined as the installers of new telephone offices, and some way will have to be found to avoid a large investment in obsolete systems.

Predictability of Performance. Predictability of performance depends upon the existence of surplus capacity and a sufficient number of simultaneous users so that no single user's work will require a substantial portion of the capability of the system. Today, the number of users of **multiple-access** computer systems is so limited that even an isolated individual will have an effect; compare the few thousand customers of a computer system with the few million of a reasonably large electric company. Compare the minimal surplus capacity in most computer systems with the thousands of megawatts held in reserve in, say, the generating system of the United Kingdom—remembering that the comparative basis is not just one of proportionality but also one related to the demands of individuals. In this area there is hope, however, of progress. Prices of processing units are rapidly falling, and it should become possible to hold adequate reserves without

incurring short-term economic problems. It is not yet so clear that this will also be possible for mechanical components such as disk stores for files, but the trends are favorable.

These points are clearly related to the matter of easy capacity enhancement. Only if components subject to capacity strain can be easily augmented will load reserve be maintained. To some extent this is possible with most current system designs. **However**, if we consider a computer to be an assemblage of processors, memories, channels, and peripherals, with as much mutual interconnection as required, then eventually a computer utility will come to consist of more than one CPU. Unless, as seems unlikely, the structure of computers becomes stable over longer periods than is now usual, the **differences** between early and late models will cause problems in the service given. Again, the analogy with other utilities is strained: The 1463 and 1973 electric generator models produce very similar 60-cycle alternating current, whereas 1963 and 1973 computer facilities are vastly different.

State-of-the-Art. Where are we in relation to the status of computer utilities as compared to other utilities? Early time-sharing systems, of a **type** intended to have a community of users rather than a mere collection (the M.I.T. Compatible **Time-Sharing** System (CTSS), the Cambridge Multiple-Access System), made a good start in this direction. The more recent developments of Multics (**M.I.T.—Honeywell**) place it as near as any existing system to **actually** being labeled a computer utility. Problems of scale still remain, however, and it is not quite clear when or whether market forces will promote the developments needed. There is, after all, competition: Minicomputers are becoming very cheap, and it may turn out that this low-cost computing power will seem more attractive than the higher cost of sharing programs, data, and power, the central theme of a computer utility. It would be contrary to experience in other developments in industrial society that the minicomputer would win out (rather like a cottage industry supplanting U.S. Steel), but it could happen.

REFERENCES

- 1969. Lampson, B. W. "Dynamic Protection Structures," *AFIPS Conf. Proc.*, Vol. 35.
- 1972. Organick, E. I. *The Multics System: An Examination of Its Structure*. Cambridge, Mass.: M.I.T. Press.

1972. Wilkes, M. V. *Time Sharing Computer Systems*, 2d ed. New York: American Elsevier.

R. M. NEEDHAM

COMPUTERS. See ANALOG COMPUTERS; COMMUNICATION AND COMPUTERS; DIGITAL COMPUTERS; HYBRID COMPUTERS; MICROCOMPUTER; MINICOMPUTERS; SPECIAL-PURPOSE COMPUTERS; and SUPERCOMPUTERS.

COMPUTERS, HISTORY OF. See DIGITAL COMPUTERS: History of; and MANUFACTURERS, COMPUTER.

COMPUTERS, MAINTENANCE. See MAINTENANCE OF COMPUTERS.

COMPUTERS, MULTIPLE ADDRESS

For articles on related subjects see ADDRESSING; INDEX REGISTER; INDIRECT ADDRESS; and MACHINE INSTRUCTION SET.

For articles on related terms see CONTROL DATA CORPORATION 6000 SERIES; GENERAL REGISTER; and IBM 360-370 SERIES.

Each arithmetic instruction in a computer may be thought of as having four addresses, two *sources* for the data operands, one *destination* for the result, and the *location* of the next instruction. Functionally, therefore, we may write an arithmetic instruction as

$$F(s_1, s_2, d)l, \quad (1)$$

where F is the arithmetic function to be performed, s_1 and s_2 are the source addresses, d is the destination address, and l is the location of the next instruction. Historically, computers have differed considerably in the way in which these four addresses were expressed explicitly and implicitly (as well as on such matters as the use of base registers, index registers, and indirect addressing).

Some very early computers, notably EDVAC and SWAC, were *four-address* computers in that each of the four addresses in Expression (1) above appeared explicitly in each arithmetic instruction. In a *three-address* computer, early examples of which were NORC and CADAC, the address l of the next instruction is implicitly the next instruction in sequence. This convention is essentially universal in all modern computers; except for conditional or unconditional jump instructions, which change or may change the normal sequencing of instructions, the next instruction is the one physically following the current one in main memory.

An attractive aspect of three- and four-address computers in the early days when hardware was very expensive was that there was no need for an *accumulator* to hold operands and results. In this context it is interesting to note that the most notable current example of a three-address computer is the Control Data 6000 series, which is a multiple accumulator system in which all three addresses refer to accumulators.

There have been two kinds of *two-address* computers. One is epitomized by the IBM 650, one of the most widely used computers of the 1950s. In this computer, one of the source addresses and the destination address were both implicitly assumed to be the accumulator in the arithmetic unit. Thus, the form of an arithmetic instruction was

$$F(s_1)l.$$

Earlier versions of this type of machine included the HEC, based on work by Booth at Birkbeck College, London, and the Elliot 400 series.

On the 650 the main memory was a magnetic drum. Optimized programming required that the address l be located on the drum such that it was just coming under the reading heads when the preceding instruction execution was completed. For the programmer to arrange this himself was quite tedious; therefore automatic optimization was a feature of assemblers for the 650, of which the best known was SOAP. Such optimization normally resulted in a factor of 2 to 3 in speed over arranging the instructions sequentially on the drum.

The other type of two-address computer is best represented by the IBM 360-370 series, in which s_1 is a *general register*, s_2 is either a general register or a location in main memory, d is implicitly s_1 , and l is implicitly the next instruction in sequence. Thus, an instruction has the form

$$F(s_1, s_2).$$

COMPUTERS AND SOCIETY

The **earliest** working example of this type of machine was the Ferranti Mark I, built at Manchester University in 1950.

Until the advent of general registers, which serve in effect as multiple accumulators, *one-address* computers were the most common type. In these an instruction has the form

$$F(s_1),$$

with the accumulator serving as both s_2 and d and l being taken as the next instruction in sequence.

A. S. DOUGLAS

COMPUTERS AND SOCIETY

For articles on related subjects see **AUTOMATION; COPYRIGHTS AND PATENTS, COMPUTER ASPECTS OF; CRIME AND COMPUTER SECURITY; DATA BANK; DATA SECURITY; and PERSONNEL IN COMPUTERS.**

Any application of computers in the public sector could illustrate how computers affect the way society operates. As examples of such applications, computers are used to process social security payments, help administer hospitals, maintain lists of persons wanted by law enforcement, and aid in the drafting of legislation. In principle, the operational procedures followed in such applications stem from policies, requirements, and guidelines laid down by the agencies and institutions within which the computers are used. But any means that greatly facilitates (or inhibits) the realization of certain ends can become important in determining whether these ends will be adopted as goals. Thus, in practice, the fact that computers are such powerful instruments for information processing leads to a blurring of the distinction between cause and effect in certain situations where they are used. For example, a national policy of basing security in old age on the distribution of monthly benefits, which are calculated from lifetime earnings, can be adopted **only** when there is a prompt, reliable method of distributing payments, and a method of automatically dealing with the enormous volume of transactions generated through payrolls.

By the same token it is natural to ask whether the fact that computers make it easier to store and

disseminate information about people does not lead both private organizations and public authorities to collect more personal information and be less careful in controlling its distribution. We are thus led to examine how computers may be affecting our major institutions and social structures, not merely how effectiveness is altered by the presence of computers, but more importantly-how computers influence the goals and very nature of societal structures and groupings. The study of computers and society, then, includes an analysis of how computers influence the operation, shape, and especially the goals of **political**, social, and cultural systems. To put the subject in perspective it is necessary to recognize that computers function in the larger area of automation and that automation is only one-although a key one-of several major technological forces that are reshaping society. Other key technological developments are occurring, for example, in transportation/communications (jet air travel, space flight, cable TV, satellite transmission) and in biology/medicine (contraceptives, transplants, drug experimentation, genetic engineering).

In this article the effects of computers on three systems are considered: the labor force, the individual, and the political system. Other articles in this encyclopedia discuss the effects on the educational system, on the health care delivery system, and the legal system. For additional reading see the references at the end of this article.

Computers and Employment. To understand the relation between computers and employment, it is useful to briefly review the earlier history of automation and employment. Along with the benefits of the industrial revolution which originated in Europe during the nineteenth century, there were widespread harmful effects, including serious displacement of labor because of the introduction of machines, and concentrated urban pollution due to the uncontrolled production of waste products. The unemployment gave rise to such events as the riots of the Luddites, who in 1811 and 1812 broke into factories and smashed machines, and the writings of Samuel Butler, who in 1872 published *Erewhon*, a novel about a utopia, in which all machines were banished because only in this way could society be certain that machines would not eventually take over control.

Computers (more precisely, data processors) are the instruments by which automation is put into effect in offices. As soon as electronic data processing equipment materialized, the possibilities of technological displacement-which manual (blue

collar) workers had been experiencing for over a century—became real for clerical (white collar) workers. In 1950 Norbert Wiener published *The Human Use of Human Beings*, which was largely a plea that computers not be allowed to cause distressing unemployment.

A noticeable result of automation is that it brought about a change in the *composition* of the labor force. During the late nineteenth century and the first half of the twentieth century, as more sophisticated automatic machines were invented and put into use, there was a steady decline in the industrial countries of the number of workers engaged in farming, textile production, and mining (primary industries), but an increase in the number engaged in manufacturing (the secondary industries). This shift was accompanied and made possible by a continual increase in productivity (i.e., the output per man-hour). Thus, sufficient goods were produced in primary industries even though population increased, the demand for goods rose, and the work force declined. The increase in demand is important, for it is this factor that has allowed total employment to increase (although there have been large fluctuations in particular industries) in the face of automation and increased productivity.

Computers, which in the United States started to appear in significant numbers in the late 1950s,

have been accompanied by still another major change in the composition of the labor force. There has been a shift to what is called the post-industrial society, i.e., from blue-collar to white-collar jobs, and to service-producing (or tertiary) employment as performed by clerks, government employees, managers, sales personnel, professionals, etc. This is illustrated in Fig. 1, where it can be seen that service and white-collar employment rose from 45.3 to 61.8% between 1947 and 1971, while blue-collar workers declined from 40.7 to 34.4% of the labor force. The role of computers in this change, particularly the extent to which computers have contributed to unemployment, has been the subject of detailed studies in many of the industrialized countries of western Europe. The conclusions are very similar to those reached earlier about automation in general, and are summarized by a statement in a report published by the International Labour Office: "For various reasons, the introduction of automation in offices has thus far not resulted in a decline in the general level of employment for office workers." (See the list below for some of these reasons.)

In spite of these conclusions it is impossible to be complacent about the long-term effects of automation and computers on employment. Sometimes, as occurred in the typesetting industry, computers gain general acceptance within a very few

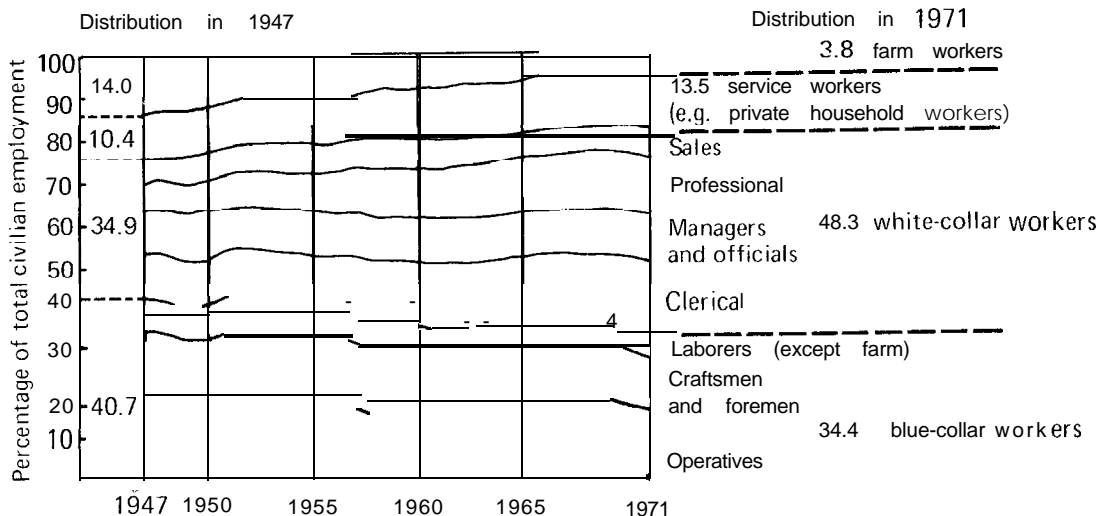


Fig. 1. Change in distribution of employment by major occupation group. Note: Statistics for 1947–1964 and 1964–1971 are based on somewhat different sets of occupational data (e.g., changes in the lower age limit for inclusion into the labor force). Similarly, the statistics for 1971 are not strictly comparable with earlier years. However, the changes have no significant effect. For 1964, the discrepancy was approximately 0.3 in the major classifications (e.g., white collar, etc.) and the two sets of statistics were averaged.

COMPUTERS AND SOCIETY

years, and the result is technological displacement and hardship for those with the specialized skills of that industry. Also, there are unanswered questions about how computers affect attitudes toward employment, in particular whether they increase the *alienation* of workers from their jobs, a major problem accompanying automation. Finally, if the increases in productivity are projected far enough (several decades), the question arises whether the decline in the work week will eventually result in greatly increased leisure time for a large fraction of the labor force and, if this does happen, whether society is equal to the challenge of making good use of increased leisure time.

During the recession that occurred in the early 1960s, when unemployment was relatively high in several countries, including the United States, some critics contended that the main obstacle to full employment was not lack of demand but rather the effect of *structural* factors brought on by automation. By this it was meant that the attributes of the labor force (education levels, skills, mobility, distribution in age, sex, etc.) were not matched to job needs. There are many examples of how automation (and computers) suddenly makes old skills obsolete and places great demands on new ones. In the United States during the early 1960s, insurance companies converted from punched card processing to computers; in the middle 1960s, airline reservation systems converted to computers, and the machine tool and process control industries started to convert; since the early 1970s, computers have become increasingly evident in banking and in hotel and travel reservation systems. Although there have been labor problems in some of these situations, for several reasons there have not been serious disruptions or much evidence of structural unemployment caused by computers. These reasons include:

1. Full automation has been attempted in only a very few cases. In most operations it has been considered necessary or advisable to have humans monitor the process and *retain* overriding control. This is partly because computerized decision making in many situations is beyond the state of the technology, and partly because of a desire to be conservative in systems design and operation.

2. Computerization usually requires years of planning, especially where service cannot be interrupted during installation. This allows time to adopt measures that will minimize the effect on employment. Both large companies and unions, because of earlier experiences with mechanization and automation, have come to recognize the need for such

measures. These include advance notice of intentions, retraining programs, early retirement plans, compensatory payments, etc.

3. Very often computers are not brought in to save labor cost, but to achieve improved accuracy, better resource utilization, or simply to do a job that cannot be done without them. In certain cases (e.g., reservation systems), humans are needed to interface with customers; in others, both office and factory, decreases in direct labor are compensated by increased needs for indirect labor (e.g., computer maintenance, keypunching, etc.)

It is an open question whether on the whole, the introduction of computers tends to produce a net gain or reduction in the skill requirements for jobs. (Skill requirements must be distinguished from educational qualifications that might be imposed for a job but which are not essential for its accomplishment.) Studies show that in certain operations (e.g., demand deposit accounting), significantly increased skill levels are needed; in others (e.g., plant operation for electric utilities), less skill is needed. A man tending a numerically controlled machine tool may have a more routine job, with less opportunity to exercise control, than he would have for the same job if it were nonautomated. In general, the net changes in skill levels (taking both direct and indirect labor into account) are not large, and the variations among industries are larger than those for the same process in different factories.

Alienation-i.e., an inability to identify with the goals and values of fellow employees and associates -has long been recognized as a serious failure of modern living. The monotony of work in automated factories is held as one of the principal contributors to this feeling, as illustrated by Charlie Chaplin's classic portrayal in *Modern Times*. Studies on whether working with computers furthers alienation are based on attempts to define and measure it. Among the contributive factors that have been identified are control (or lack of control) over the work process, the ability to recognize the purpose of the work, and the confidence that good work will bring proper recognition and fair compensation.

Comparisons between industries with and without computers, or between situations before and after a computer is installed, reveal (as with the debate on skill levels) that the question is complex, with some evidence that the alienation resulting from computers is less than that found in other automated processes. Workers in computer-related jobs see their work as requiring greater responsibility, more accuracy, and more exacting deadlines. But often they

also feel that middle supervisory positions have been eliminated and that their chances for promotion have therefore been reduced. It is a question whether such decreases in alienation will continue as the novelty and prestige of working with computers wears off. If eventually the attitudes become like those found on assembly lines, then the means being sought to reduce or compensate for boredom and alienation in those situations will also have to be applied to computer-based jobs. These measures include providing increased job benefits, redesigning the work environment so that the job becomes more interesting (this may involve putting together work teams), and developing compensating leisure activities.

Most of the people in most of the world must concentrate their full energies on acquiring the necessities of food and shelter for themselves and their families, and therefore it may seem unrealistic to be concerned with the problem of what to do with leisure time. But eventually the population of the world must stabilize; the finite limits to resources mean that the per capita consumption of material goods will also have to reach a limit, and if the rise in productivity experienced over the past century continues, man will have more and more time to spend on activities other than work. (Of course the demand for services-medical, educational, recreational, etc.-could grow without corresponding increase in requirements for material goods.)

In the United States the average work week has been decreasing about 3 hours each week every decade since 1890; at its present level it averages 37 hours. Extrapolations of the trend suggest that by the year 2000 the working year could go down to as little as 1,100 hours, which could take various forms: e.g., 27 full time (= 37.5 hours) work-weeks per year, 50 short (3 day) weeks per year, or full time work-weeks for the whole year, with retirement at age 40.

Leisure is not to be equated with time free of work-the unemployed do not have leisure, nor is leisure the time spent in queueing for recreational services or job retraining. How people spend their leisure is highly dependent on social class, educational level, financial assets, nationality, and age. Working classes tend to use their extra time in relatively unstructured ways; working around the house, watching TV, relaxing with their families; professional groups, which on the average have less leisure, are more interested in activities such as music or participation in discussions; sports and travel are of interest to all groups. Educational institutions are playing an increasingly important role, as shown by enrollments in part-time degree

programs, in nondegree courses (photography, gardening, language study). However, fundamental questions remain as to whether society can adapt to large increases in leisure time.

Some sociologists (e.g., Erich Fromm, David Reisman) feel that the work ethic is deeply ingrained, doubt whether leisure can be satisfying unless work is satisfying, and see the need to integrate the two activities within a common framework. But studies also show that industrial workers do not consider work to be their central interest, nor are their social relations based on their work. Computers, representing an advanced state in the mechanization and automation of work, are undoubtedly bringing nearer the time when large elements of society will have to come to grips with the problems of increased leisure. The answers to these problems will have to be found within a very broad context involving the structure and goals of society.

Computers and the Individual. Until quite recently, computers were expensive devices, accessible mainly to large corporations, governments, and institutions. In important ways, their use affects the lives of individuals. The effect that has been the subject of the most intense inquiry is the relation between computers and privacy. Interest in this stemmed from a series of US. congressional hearings on the gathering of information about private individuals. The hearings revealed that-aided by newly developed electronic devices for eavesdropping, surveillance, and personality testing-many agencies, both private and public, had enormously increased their activities in gathering personal information about employees and competitors. (Eventually, laws were passed to limit such surveillance and wire tapping.) When in 1965 the Bureau of the Budget proposed establishing a Federal Data Center to preserve economic data, this was viewed as a threat to individual freedom. Although the proposal was rejected, the Congressional hearings initiated (in the United States and subsequently in many other countries) a fundamental review of how data banks containing personal records (e.g., on credit, health, crime, educational performance), and more particularly computerized data banks, affect the rights of individuals. The concerns centered on a number of points:

1. Often, individuals are not aware that information about them is being collected. Even when they know about the existence of an information file, only very rarely is there an opportunity to see the records, challenge misrepresentations, and have mis-

COMPUTERS AND SOCIETY

takes corrected. These practices persist in spite of the fact that records gathered for insurance, financial, or medical purposes can contain opinions based only on hearsay, and in spite of frequent errors with many computerized billing systems.

2. The existence of the computer and the availability of mass storage encourage the collection of unnecessary data, particularly by governments. Further, there is the feeling that the computer is unforgetting and unforgiving, and instances of youthful indiscretions, misdemeanors, and delinquent payments are kept for times much longer than can be justified.

3. Records about an individual are passed about too freely. The existence of terminals and remote job entry makes it easy to disseminate medical reports, credit ratings, court actions, and school records, and the computerized files on which such records are kept are not adequately protected against unauthorized access.

Although measures have been put forward in some jurisdictions about certain types of information processing in response to these concerns, there are many reasons why progress is slow. First, it is very difficult, especially in countries where the common law is the basis of the legal system, to define a general right to privacy. Such a right inevitably conflicts to some degree with freedom of speech and freedom of the press, two very jealously guarded freedoms. Thus, in 1967, the Freedom of Information Act (U.S. Code, V15, Para. 552; suppl. III 19651967) was passed in the United States to insure access of information that should be open in the public interest.

Second, personal information is essential for the conduct of many of the functions of government and business. Government must collect such information, for example, to issue driver's licenses and welfare payments or a license to operate a prepaid medical scheme; businesses are entitled to data that allows them to establish credit ratings. Even more data is needed for planning (e.g., schools or housing developments) and for research, particularly in the social sciences.

Third, it is difficult to distinguish legitimate needs for information from excessive demands for unrequired data. There is a legal requirement to provide census data, but for every census it has been necessary to review what should be collected. Consent of the person providing the data is a useful concept, but it is not always easy to decide when consent is freely given (consider the case of a student or a welfare recipient). In any case, certain infor-

mation should be regarded as matters of public record.

Fourth, as noted above, the concept of privacy is so strongly intertwined with other legal rights that any laws about it have to be carefully built into the existing legal framework. In countries where there are federal systems (the United States, Canada, Australia, West Germany), this can be particularly difficult because jurisdiction is divided between the levels of government. Moreover, when laws are passed, it is necessary to recognize that computerized data banks are not the only repository of personal information. If protection is to be effective, all other, more traditional, systems must also be considered.

These difficulties notwithstanding, a number of legal steps have been taken to strengthen the rights of privacy in the presence of the computer. Most common has been the enactment of laws regarding credit data banks (e.g., in the United States, the Fair Credit Reporting Act, 1969). These laws have usually established the right of persons to see their records, and have required mechanisms for correcting mistakes. Generally, it has been recognized that the security of data is a responsibility quite distinct from the protection of privacy. To an increasing degree, computer systems are incorporating both hardware and software devices to insure that unauthorized persons cannot gain access to confidential data. In some places (e.g., the State of Hesse in Germany), an ombudsman has been appointed to rule on disputes where computerized data banks are involved; in 1973, Sweden adopted a law that had been proposed in many places, one requiring that data banks containing personal information be licensed and that they follow laid-down procedures in their operations.

What has failed to emerge is the definition of a general right of privacy based on the notion that an individual has certain rights or ownership of the data about himself. Even more important is the slow progress in making a right to privacy effective (except, perhaps, in the case of credit data banks). The most difficult areas are undoubtedly those involving national security or crime records, for in these cases exceptions have to be made in relaxing certain safeguards that might be mandatory in other situations (e.g., not recording hearsay information).

The inquiries, debates, and studies about computers and privacy are still continuing (NAS, 1972). There is some consensus that progress will have to be made along the lines of working out procedures appropriate to each of the areas for which personal records are kept (employment, health, education, police). Most important is the recognition that data bank issues are not primarily technological, but

rather administrative, political, and legal, and that they arise out of changes in the relationships among individuals and organizations. For these reasons, the main protective measures will have to be based on those long-established legal procedures or "due processes" on which all individual rights are founded: well-defined practices, rulings in open courts, and opportunities for challenging and reviewing decisions.

Quite aside from the issue of privacy are other aspects of the relationships among computers and individuals, which are being carefully reexamined. As computers become more and more used in administrative processing, questions arise whether (from the point of view of the individual) computer-based procedures are better or worse than the earlier manually based systems. Computers are installed for a variety of reasons, the most important of which are to achieve economy of operation and resource management, to speed up procedures, and to carry out certain operations that could not be done otherwise. It does not automatically follow that benefits accrue to the individual who is involved in the procedures. Is a computerized system of paying insurance claims, for example, as fair to an individual as one in which he deals exclusively with people? Is the claims form simpler? Is there equal opportunity for appealing a ruling? Is payment as fast? Is he made aware of all the consequences of presenting a claim?

As computers become more and more involved in the dealing between individuals on one hand, and governments and business on the other, it becomes increasingly important that there be answers to such questions. There are aspects of systems that operate even beyond fairness and promptness. Can systems be made to respond in a "human" rather than machinelike fashion? The properties of a system that seems human, or humane, are not altogether clear, but one might look for politeness, an ability to deal with special cases, and a value placed on satisfying the people involved. Human bureaucracies have long predated computers, and world history and world literature are full of instances of societies where rulers and administrators have had no regard for the sensitivities of those who live under their rule. But the question is: Are present-day computer based administrative procedures better or worse (from the point of view of the individual) than the systems they have replaced, and are computerized systems as satisfactory as they can be? As a particular subquestion, how much human presence is needed in a computer-based system so that it will be an acceptable substitute for personal relationship?

These questions are only beginning to be asked, and the responses have not yet influenced the design of new systems. But the more broadly phrased question, whether technology is benefiting the individual, has been asked for some time. In fact, this is the dominant theme of modern inquiry, and there is a wide divergence in the answers and beliefs of those who have addressed themselves to it.

There is an articulate school of technology critics (among them are Herbert Marcuse, Lewis Mumford, and Jacques Ellul), whose adherents feel that the industrialization and urbanization that have resulted from technology have turned out to be disasters. They reason that technological growth has become an end in itself, altogether removed from its original purpose of providing man with the goods and services to better himself socially. They see man as having become preoccupied with techniques and toolmaking, and in so doing has lost sight of humanistic values. The computer is regarded as an especially serious threat, for its use depends on factors such as calculability, precision, and efficiency and these essentials necessarily dominate.

Others who have attempted to assess the role of technology in society have also seen the influence on human values as detrimental, but they are less pessimistic of the relation and are more ready to believe that control can be exercised over technological developments (Erich Fromm, Jacob Bronowski, Marshall McLuhan, and Alvin Toffler are in this group). Recognizing that the rapid rate of technological change has led to challenges to the authority of religion, the family, and the state, as well as simultaneous desire for security and stability on the one hand and for innovation and mobility on the other, they do not concede that such value conflicts are necessarily bad. They argue that good planning can lead to an integration of conflicting values and that technology, far from being inconsistent with human goals, can lead to reemphasis of these goals.

If we interpret this general argument in terms of computers, it is obvious that computers can be used to let individuals choose color and style when purchasing a car, or to find a donor with the right type of blood. But can they help in allowing people to make important choices in life (e.g., in determining how and by whom they are governed, in choosing what they work at or where they live)? Can they help in reducing the universal threats of war, pollution, and overpopulation? When approached in these terms, it is obvious that computers represent but one set of forces acting upon individuals. Depending on one's philosophical outlook and degree of optimism,

COMPUTERS AND SOCIETY

either one of two philosophies is acceptable: One can believe (1) that computers represent the latest of the technological forces acting to reduce choice and limit the activities of human beings; or (2) computers offer new possibilities in dealing with large populations so that attention to individual needs is a practicability and not just an ideal.

Computers in Organizational Structures and in the Political Process. Individuals and societal groups are the limiting members of a large number of organizational units that constitute society and interact with each other in various ways. As other examples of units, we might consider a government that comprises its electorate; the different entities of government at municipal, state, and federal levels; the functional departments within a company in the private sector; and citizen-action groups interested in the environment or civil liberties. Each unit arrives at a size, creates its own *modus operandi*, and acquires a sphere of influence that depends on its function and which determines its effectiveness. Computers today are altering the effectiveness of these different units and are leading to changes in the balance of powers or political relationships among them.

Computers play a role in determining the balance of powers because they are such powerful instruments for gathering, organizing, and disseminating information. Sociologists and political scientists have come to recognize that information is a new source of power, complementing the traditional sources of land and capital. The possession, distribution, and utilization of information is seen as a new major activity comprising what has been called the "knowledge industries." Many people are engaged in these industries: scientists, engineers, mathematicians, librarians, teachers, reporters, planners, management consultants, etc. Computers and computer personnel form only a part of the information-gathering activity, but they are an important part because information processing is the central function of a computer and many participants in knowledge industries use computers in their respective specialties. The fact that computers are used by librarians for information retrieval, by teachers in computer-assisted instruction, by lawyers looking for a legal precedent, or by politicians compiling lists of possible campaign donors is, of course, important, for all of these activities illustrate how the effectiveness of information gathering is enhanced. Beyond this direct benefit are indirect effects whereby the presence of computers in some way alters the climate in which things are happening. As already

demonstrated in the case of employment and privacy, these indirect or secondary effects may, in the long run, be more important than the primary ones.

One question that might be asked about the political effects of computers is whether those who work with computers (programmers, analysts, system designers) have acquired an unusual or undue amount of power because of their expertise. The answer seems to be that they have not. One reason is that the politicians, administrators, and civil servants who exercise power can do so only by learning to make use of specialists like lawyers, engineers, and accountants; computer experts represent one more group that has to be managed. Another reason is that, in spite of promises and plans, there has been a noticeable lack of success in producing management decisions by computerized systems or in system design of complicated operations involving many people and functions. The result is that there has been no usurpation of management authority or decision-making powers, either by computers or by the people who work with computers. This situation is similar to the position of United States scientists in the decades of the 1950s and 1960s. In spite of very heavy concentration on (and expenditure for) electronics, nuclear technology, and space programs, an examination of the record shows that the key decisions were made by politicians on political grounds, acting at times on and at other times against the advice given by technical experts.

Large organizations exhibit structure, and of the many possible structural forms two basic ones can be identified. The first is a *centralized* mode in which each subunit (except the headquarters) reports to a single unit above it in the hierarchy. This corresponds to the mathematical concept of a tree, with the headquarters as root. The second structural mode is *decentralized* in that each subunit may be connected to any of or all other units. This corresponds to a graph. For any complex organization in the public or private sector, decisions have to be made, regardless of whether the centralized or decentralized mode dominates, and there may be a continual shift between the powers of the headquarters and those of the subordinates. There are extensive arguments to support either mode of structure, but basically the current philosophy is that centralization allows a better overview and better control of the whole system and offers better opportunities for taking advantage of "economies of scale"-the economic thesis that it is cheaper to manufacture things in large quantities. By contrast, decentralized organizations have an advantage in that control of the operation is closer to the end-user,

so that service is likely to be faster and more responsive to user needs.

We can now ask the question whether computers favor or discourage centralized control. Early computers were large and expensive, and large-scale operations were usually needed to justify an installation, so that in this sense they favored centralization. More recently, as costs of hardware have come down, and time-sharing, remote job entry, and computer communications have become more common, it is becoming increasingly economic to run a decentralized computer system. This means that computer systems can be designed for either centralized or decentralized organization, and the decision on which structure to adopt (or which mode is to dominate) can be made on the basis of management philosophy rather than purely technical considerations. The trend to computer networks and distributed data bases is in effect influencing a trend toward decentralized organizational structures.

The factor that is becoming centrally important in determining how computers redistribute power is *access* to computer data and understanding of what can be done with computerized information systems. This is not quite the same thing as ownership of a computer. There probably was a time when ownership of a computer center conferred a certain amount of authority and prestige to its owner; for example, a government department that ran a computer center could formulate plans and influence decisions because it had sole access to special expertise. But with the great increase in the number of computers, the emergence of computer utilities, and the growth of minicomputers and time sharing, there are no marked advantages to be had from owning hardware. (There may, in fact, be some disadvantages because it is more difficult to adapt to new technology.) What benefits there are from computers, are derived from building up data bases along with systems and routines for extracting and processing the data, preparing reports and plans, and being able to present the results effectively.

Many in the industry feel that, to maintain a balance of power, computers must be made easily accessible not only to governments and large organizations, but to small unstructured groups and to individuals as well. But it is not enough to place a computer at the disposal of a community group, a body of students, or a citizens' action committee. If the computer is to be used for a purpose other than routine accounting and data processing, it is necessary to build up software and especially large data bases on broad subjects such as housing, recreational facilities, zoning by-laws, and employment registries.

Systems of this type are not only expensive, but are also close to the limit of present-day technical capability, especially if they are expected to respond to general queries in a language close to English, a desirable feature if they are to be used by untrained persons.

The most interesting possibilities for broadening the accessibility to computer-based systems are anticipated from developments in cable TV and computer communications. It is technically feasible to have terminals that can accept data from the home and be switched to different receiving centers. Based on this, a great many specialized services could be provided in the home-message exchange: shopping; meter reading; computer-assisted instruction; participation in surveys, debates, voting, etc. The home terminal would act as a localized access point to service centers and to information centers such as schools and libraries. At the same time, the system would augment and extend the functions of establishing contacts between groups and disseminating ideas and proposals, functions now carried out to a large extent by newspapers, radio, and commercial TV.

How multiple-purpose, two-way computer communications will evolve, depends among other things on economic considerations of whether a fair rate structure will be offered to users, manufacturers, purveyors of services, and to the general public. If the systems can be even partly realized, they could introduce a new major technological force for change in society. Some have expressed concern about the instabilities that might result if too rapid a response rate were built into the political system. If it were easy to conduct nationwide polls, what effect would this have when legislation on the elimination of capital punishment was being considered? Elected politicians have dual roles: They are expected to act in accord with the wishes of their electorates, but they are also supposedly chosen to represent their constituency in accord with their judgment and experience, which requires them to vote on issues after carefully weighing all the relevant arguments.

The question of political responsiveness is another illustration of the employment and privacy quandary. Computers, in common with other major technological developments have the effect of heightening certain issues that confront society. In some cases (e.g., centralization versus decentralization), computers can probably be used with equal effect to achieve either of two opposing goals. In other cases (e.g., employment), if it is desired to realize the economic benefits of using computers, the secondary effects are less manageable. In all of the many places

COMPUTING CENTER

where computers affect issues, technological changes are forcing reexamination of values and goals. The success in resolving the issues will be dependent on the success in integrating value conflicts.

REFERENCES

1950. Wiener, Norbert. *The Human Use of Human Beings*. Boston: Houghton Mifflin.
1968. International Labour Office. *Labour and Automation*, Bull.7. Geneva.
1971. Westin, A. (Ed.). *Information Technology in a Democracy*. Cambridge, Mass. : Harvard University Press.
1972. National Academy of Sciences. *Data Banks in a Free Society*. New York: Quadrangle Books.
1973. Gotlieb, C. C., and A. Borodin. Social Issues in *Computing*. New York: Academic Press.
1973. Parkman, R. The *Cybernetic Society*. New York: Pergamon.
1973. U.S. Department of Health, Education, and Welfare, Secretary's Advisory Committee on Automated Personal Data Systems. *Records, Computers, and the Rights of Citizens*, DHEW Publ. No. (05). Washington, D.C.: U.S. Government Superintendent of Documents, pp. 73-94.

C. C. GOTLIEB

COMPUTING CENTER

For articles on related subjects see **APPLICATIONS PROGRAMMING; DATA PROCESSING; DATA SECURITY; OPEN SHOP; OPERATING SYSTEMS; PROCESSING MODES; PROGRAM LIBRARIES; SECURITY OF COMPUTER INSTALLATIONS, PHYSICAL; SERVICE BUREAUS, DATA PROCESSING; and SYSTEMS PROGRAMMING.**

For articles on related terms see **ADMINISTRATIVE-BUSINESS APPLICATIONS; FILES; MULTIPROGRAMMING; and SPOOLING.**

A computing center provides computer services to a variety of users through the operation of computer and auxiliary hardware, and through ancillary services provided by its staff. A not-for-profit service center usually provides such services *at cost* to its *users*; but a for-profit center does so with the

intention, at least, of making a profit from providing services to its *customers*. Since there is little distinction between the two centers other than pricing and the label used for the consumers of their services, we will deal with the computing center as a service center with users.

Services. The extent to which the users participate in the operation of the center determines its staffing and organization. At one extreme is the completely **closed shop**, wherein the users supply only the initial specifications for the computations to be carried out or the reports to be provided. Thereafter, they supply only the new data and/or current information used to bring files up to date and satisfy requests to supply computed output or reports according to those specifications. This closed-shop organization requires that all skills and equipment required to provide its services be supplied by the computing center.

At the other extreme is the completely **open shop**, wherein the users supply the initial specifications, convert them into computer programs, supply the current information, convert it into machine-readable form, and operate the computer to obtain the desired results. This arrangement requires only that the computing center supply the computer, the manuals on how to use it, some supplies, and heat and light. Most centers today are nearer the former than the latter, but many maintain closed-shop machine' operation while providing both open- and closed-shop programming, as mentioned below.

Fundamentally, there are three services provided by a computing center, each usually offered with some degree of open-shop organization. The most obvious service is **machine operation**. This skill is not learned in a few minutes for any computer, and is acquired only after lengthy training on large sophisticated machines. For this and a variety of other reasons, as explained later, all but the smallest computing centers offer only closed-shop machine operation for the majority of their operating hours. Some provide open-shop (or partial open shop) operation during restricted hours or on weekends, but these are in the clear minority.

Perhaps the next most obvious service required is **programming and system analysis**. This service is most easily and conveniently provided on both an open- and closed-shop basis. Those users who wish to do their own analyses or write their own programs may do so, and those who wish to have them designed and written by the computing center staff (and can afford to pay for this service) may also be satisfied. Since (from the computing center's point of

view) there are two kinds of programming, those services are usually divided into two departments: systems programming and applications programming.

The least obvious service provided is that of data control, scheduling, and quality control. This service deals with the inspection of the incoming information for completeness and timeliness, with scheduling the machine operation for applications whose current information and files are ready for processing, and with inspection and dispatching of output reports and updated files from applications that have been processed.

MACHINE OPERATION. In the smallest computing centers, or otherwise in those which provide the greatest degree of open-shop programming, open-shop machine operation is fairly common. Frequently, a machine operator is available to run the machine for some applications, to train prospective open-shop operators, and to assist with the machine operation of some aspects of other applications. Almost inevitably, however, as the work load of the center increases, open-shop machine operation is the first service to be restricted or eliminated.

One of the most compelling reasons for a restriction in open-shop operation is the security of the data files. As long as data files are small or easily replaced, little attention needs to be paid to their security. When files become large enough to be stored on some medium that must be kept at the computing center, however, the possibility that they could be destroyed or inspected (either inadvertently or deliberately) by another user arises. To the extent that either of the above is undesirable or costly, the cost of closed-shop operation can (and almost always does) become more attractive to the center owners.

As might be expected, open-shop operation frequently gives rise to inefficient use of the computer. In a new installation where there are not sufficient applications yet programmed, such inefficiency is of no particular moment. However, even if file security does not materialize as a problem, inefficient use of the computer may compel the introduction of closed-shop operation to escape the alternative requirement of a faster machine. In such cases the cost of operators is almost always the much more attractive alternative.

Machine operation varies in the amount of skill required, from nominal on small machines with simple or nonexistent operating systems to very substantial on large, fast machines with sophisticated operating systems. Since the operating system is

supposed to speed up operation of the machine by providing a smooth transition from one application to the next (called "job-to-job transition"), operators must be trained in the language and procedures of the operating system.

The operating system and the operator must communicate about the running of some applications. This is accomplished via the operator's console, usually a display screen with a keyboard or a typewriter (most commonly the latter). The operating system sends messages to the computer, indicating operator intervention is required. For example, if the card reader jams or tears a card, thus inhibiting its action, it usually indicates a "turned-off" state to the operating system when the system issues a request to read a card. Upon noting the card reader in a turned-off state (either due to a jam or a partial power failure, or whatever), the operating system would issue a message to the operator, indicating the condition. Once the problem had been cleared, the operator would need to know what characters should be keyed into the console to indicate the back-to-normal condition.

Of course, in very simple computers, the communication is much more simple-minded. The computer issues the request to read a card and then just waits until the information is transmitted. If the card reader is jammed or turned off, then everything comes to a standstill until the problem is rectified. The difficulty with this situation is that some training is required for the operator to notice that the machine is waiting longer than usual for some operation, and then to know where to look to find out what it is waiting for, and finally how to fix it. Since the largest and fastest machines can perform millions of operations per second, their operating systems will typically be designed to note the fault, issue a message to the operator, and carry on with whatever can be done without the use of the faulty component.

SYSTEMS PROGRAMMING. Systems programming deals with the writing and maintenance of programs that are part of the computer operating system. The amount of systems programming skill required in a computing center is dependent probably as much on its management philosophy as on its size (an obvious factor). Most general-purpose computers are made available by their manufacturers complete with operating systems. The earliest machines had none or only very rudimentary operating systems, whereas modern machines have very sophisticated operating systems. These operating systems (or just systems) are designed for use by a typical installation and provide for installation-set parameters to be varied

COMPUTING CENTER

to meet a given installation's needs.

For example, one parameter in most operating systems is the number of files that will be maintained on permanent mass-storage media (e.g., rotating magnetic disks). This parameter is important because space must be allocated to catalog all the attributes of each file (such as its name and number of records it contains). Since these are permanent files, these names must be stored somewhere for ready access by the operating system. It will not do, for instance, to store the catalog of file names on a tape that the operator must fetch and mount every time one of these files is referred to or altered. On the contrary, this catalog (often called the "file name table") must be kept in main memory or on a mass-storage device. The point is that some large installations with very large numbers of files will have to allocate much space to store file-name tables, whereas a small installation will not wish to tie up a lot of valuable space for only a small file-name table.

Many such installation parameters are set to help tailor the operating system to fit a variety of needs. Often, however, the operating system, even with all of its parameters set, still falls short of the installation's needs. The usual case is that it can meet most needs, but meets some critical need only marginally or with low efficiency. For example, a computing center whose purpose is to run applications that simulate nuclear reactors will typically run a few very long jobs (on the order of hours of running time each) in a day. On the other hand, a programming school might run a very large number of very small jobs (each taking only a second or two to run). Even with a well-designed operating system, its machine accounting functions (to keep track of which jobs used what amounts of computer time) are unlikely to be adequate for both installations. In such a case the installation management must decide between the costs of inadequate or inefficient operating system performance in the accounting area, and the costs of systems programming talent to modify the operating system to meet its specific needs.

This decision is not nearly as simple to make as it seems on the surface. Since the computer manufacturer provides the operating system with the computer, and since such systems are made up of very sophisticated programs (even for relatively rudimentary systems), they are almost never fully debugged. Accordingly, the manufacturer provides software support (or operating system maintenance) to fix the bugs as they crop up.

Since the manufacturer has a support group of

systems programmers, he is the target of many requests for improvements and enhancements to parts of the system that perform their published tasks properly. These requests for improvements come from the installations using the equipment and from the manufacturer's own sales organization, which recognizes that it could sell more machines if the operating system had more or better capability in certain areas. Regardless, if an installation does not make any changes in the operating system it receives from the manufacturer, it can expect a much more sympathetic hearing if the system supposedly fails to perform in some area. Just as in manufactured goods, the manufacturer feels much less compelled to support a device that has been "tampered" with (even by competent people) than he would for one that is still in its delivery state.

Principally for this reason, systems programming tends to be an all-or-nothing proposition. Either a shop has no systems programming talent or it has enough to become completely familiar with and substantially provide overall support for its operating systems.

The argument for no systems programming talent is that the costs of inefficiency or incapacity in some areas are less than the costs of learning about and maintaining an operating system. The opposite point of view is that if the operating system needs work that the manufacturer is not inclined to supply, then work on many marginal areas may as well be done too. This is usually the policy of the larger shops with specialized work loads not encountered by most users of the equipment. Systems programming, because of the relatively high level of sophistication of the programs, is usually staffed with the more experienced programmers. For this reason the systems programming function often serves as a consulting function to the applications programming staff, as well as performing the functions mentioned above.

APPLICATIONS PROGRAMMING. Applications programming is concerned with the writing and maintenance of programs that accept as input the information supplied by the users and possibly combine it with information on file to produce output for the user. In that context, applications programming is at the heart of the purpose of a computing center: making machines do what people want. As mentioned above, this service is the one most likely to be a mixture of open and closed shop. In a bank, for example, it is very likely to be a closed-shop function. The people for whom reports will be written will not be expected to have much programming proficiency. In an engineering or re-

search department or at a university, however, frequently the users will have had computer programming training. This, combined with the nature of the reports, often results in more open-shop programming in the latter than in the former.

Frequently, the nature of the output is all-important, however, and mandates a closed-shop approach. For example, a report to be prepared for reading by many users on the basis of data submitted by many users will not usually be a good candidate for open-shop programming. Such a report should probably be programmed by some central function (such as the computing center) with the needs and interests of all users in mind. On the other hand, a program to calculate some pump-flow rates for the only piping engineer in the department would probably be a good candidate to be written under open-shop programming conditions. In that case, since the engineer would likely be the only user, the instructions on how to prepare the input data might never be formalized (i.e., documented), a frequent result in open-shop programming environments.

Sometimes, the nature of the programming itself dictates the need for an open or closed-shop applications programming approach. Simple programs usually have still simpler specifications. Often, however, very complex experimental procedures, for example, can be specified only to the extent that they can be coded into programs. In such cases, especially when the procedures are not well defined, the trained user himself can code the programs about as easily as he can specify exactly what must be done. In that case, one whole step-fraught with communications problems and potential errors-can be eliminated from the programming process by using an open-shop arrangement.

DATA CONTROL, QUALITY CONTROL AND SCHEDULING. Once again, the extent of this service is determined by the extent of open-shop practices at the installation. Clearly, in a fully open-shop (programming and operating) arrangement, users will provide their own data, validate and control it themselves, schedule their use of the machine to coincide with the availability of the latest data, and check their own reports. In that case, no service of this sort needs to be offered by the computing center. On the other hand, in a fully closed shop, a great deal of data handling prior to the production run will often be required. Thereafter, if the reporting system is complex, or if many reports are routed to several destinations, staff must be provided by the computing center to handle all those chores.

For example, in an application where man-hours are accumulated and posted for a department

each week, somebody has to verify that each man submitted a time card. **All** the time cards must be checked to see that employee numbers are correct, that legitimate charge numbers were used, and that the hours **charged** are reasonable. Some of this checking can be done using computer programs that compare those numbers to sets of numbers on file. **But** that does not verify that a man used the correct numbers, only that he used legitimate numbers-numbers that are permissible to use. Some parts of the checking are best done in the department. For instance, **in** a department with large fluctuations in **manpower**, somebody familiar with everyone present **might** most efficiently check that a time card was collected from each man.

Keeping track of what has been received and processed by the computing center can be done only by the computing center staff. This service may be divided into two categories: checking that the center processes all the data it receives and checking that it receives all the data sent by the using department. Obviously, it will do little good to check that every man in a department submits a time card if the computing center cannot determine if it got all the time cards. Accordingly, much of the checking about amounts of data is handled by both the user and by the computing center. Then, before a production run is made, either on a special or periodic schedule, the user and computing center reconcile their separate control records to be sure there is agreement. Frequently, for example, in an application such as the time-card system, both the center and the user keep a written record with batch numbers, numbers of time cards, and total man-hours in each batch. The user counts the cards and totals the hours manually before sending the time cards to the computing center for processing. Upon receipt of the cards, the computing center punches the information into tab cards or some other machine-readable form. Such recording equipment sometimes accumulates card counts and total hours as the data is recorded. More commonly, a special computer program is used to read each batch of time cards, count the number of entries, total the hours, and verify that legitimate numbers are used, etc. Inspection of the counts and totals verifies (or contradicts) the manual counts and totals sent by the user. Once these are reconciled, the production run can be made.

Concern that all the information received is properly processed lies exclusively with the computing center. Typically, when each batch is added to the master file, a report is generated to display the total number of entries and total number of hours (using again the preceding example) on file at the

start of the **run, added** as a result of the run, and on file ξ the end of the run. In addition to file-labeling and checking by the programs, a manual record is frequently kept to show counts and totals before and after every run made to add time cards to the file. In that way, in the case of reruns, when file-label checking sometimes needs to be bypassed, files can still be checked for completeness. Curiously, one of the biggest headaches in the data control area is not in making sure that all data has been added to the file, but that it has not been added more than once. This occurs most frequently when a file is updated with bad information, requiring a rerun.

Since the references in all the programs are to the latest file when one is updated, care must be taken in the case of a rerun to update the **second-from-latest** file. Further care must be taken to destroy the former latest file so it will not be confused with the one produced by the rerun. Every installation manager has aged visibly when, while walking through the machine room in such a situation involving novice operators, he sees two files, both labeled "latest." For these reasons, several generations of files are kept on hand at all times: the latest, the one before that (from which the latest was made), the one before that, and so on. Typically, four such copies are kept as backup to the latest file, but fewer are kept in applications not prone to error, and more are kept (up to six or eight) on applications subject to high error rates or many updates in a short time period. The shorter the period, the more backups are required. For example, three update runs, all in the same shift, all performed by the same operator may be handled incorrectly. If all the backups were created using the same bad technique, the file would be in danger of being wiped out (destroyed or rendered useless for purposes of making an update run).

Notice that each time a file is brought up to date (or updated), an entirely new copy of the file is made, in which the new information has been added. At the end of such an update run, the old file is intact, exactly as it was prior to the run, and the new copy contains everything from the preceding file plus all the new material. On the surface, this may seem extravagant, especially when compared to manual file maintenance. Imagine the cost to copy an entire file of letters everytime a new one was added to the file! But it would certainly insure that there would be adequate copies of the correspondence.

In computing, the cost of updating is nearly negligible by contrast. If the file is kept on tape, for example, a new copy can be made as each record is read, in substantially the same time as that required

to simply read the file. Since there are always at least two tape drives available, the updated tape can contain a copy of the preceding generation of the file at substantially zero cost. This technique insures against operator, equipment, and program failure, since it allows for reproducibility of any update run. All that is needed to re-create any edition (or generation) of the file is the previous edition and the new material that was added. Contrast that with the case where the new edition is created by reading the old edition up to the end and then adding the new material to the end of the old file, thus making it the new one. Such a practice is not reproducible because the "end" of the old information is no longer identifiable. Accepted practice is to keep enough previous generations of files on hand so that any operator, equipment, or program error can be detected and corrected (by running the reproducible run that re-creates the faulty generation), all before either the backup or current information is discarded.

Occasionally, of course, errors are not detected in time, especially programming **errors** that introduce subtle errors into the files at each update. In such cases, files need to be regenerated from scratch. For this reason, current data is often stored for very long periods of times. Such a practice is called "archiving," wherein either the cards themselves, microfilm images, or separate files on magnetic tape are stored in case it is necessary to go back to a version of the file beyond the usual backup period. To reduce the incidence of having to regenerate a file from scratch, sometimes year-end or quarter-end copies are stored separate from the usual backups. In addition, hard-copy reports are sometimes used as a starting point in the case where all file information is destroyed. At any rate, even though the data control function is supposed to keep these problems from occurring in the first place, it must be aware of how best to detect and recoup any foul-ups well before the last good copy of the file is retired.

Kinds of Computing Centers. Computing centers provide service to a variety of constituents, using a variety of equipment and personnel configurations. We have already seen that the degree to which open-shop practice is allowed determines to a large extent the amount and kinds of services supplied by a center. The character of the work load also determines what levels of which services must be provided, but to a relatively smaller extent than the degree of open-shop practice. Some work loads lend themselves much more easily to a high degree of open shop than do others. Principally, computing

work load (and hence the installations that cater to them) can be categorized into two categories: batch (including remote job entry) and time sharing. Historically, the first general-purpose computers were batch machines. In that kind of computing, jobs are processed in serial fashion, one after the other. Each job has exclusive use of the computer and all its peripheral devices (card reader, punch, magnetic-tape drives, disk drives, etc.) during the time it is being executed.

Until the mid-1960s, batch was by far the most common arrangement. But even the most casual observer will note that a batch arrangement makes very inefficient use of some resources most of the time. Consider, for example, the small job consisting of a program and some data on cards that produce a report on the printer, which is simply an exact listing of the information in the cards. Such a program is called a "card-to-print program," and if it is generalized to provide headings and optional spacing and other information, it is called a "card-to-print utility." At any rate, such a program does not use either tape or disk storage equipment. If the listing is short, of course, storage is not of much consequence, but if the listing is long, the tape and disk equipment are wasted for a long period of time. Similarly, the main memory of most machines is much larger and much faster than required for a card-to-print application, and presumably the arithmetic unit would be used only for counting line spacing, for page numbering, etc. Therefore, as faster computers were installed, ways had to be found to improve the efficiency of the computer use.

One fairly early solution was tape-oriented batch systems with two computers: the main computer and a smaller machine. All tape drives in such a system are wired to the main computer. At least one, and sometimes two, of them are also switchable to the smaller machine, to which the card reader, punch, and printer are also attached. Input jobs are loaded onto tape by the smaller machine from cards. The tape is then rewound and switched to the main computer, which is programmed to read that tape for all card input. Another tape drive is used by the main machine to write out all output destined for the printer and the card punch. After a batch of jobs has been run, that output tape is rewound and switched to the smaller machine to be read and listed and/or punched. Using this technique, processing times on the main computer are speeded up, but time is required to load a batch of jobs to tape. Of course, the first job cannot be started until the last job is loaded on and the tape rewound and switched to the main computer.

This apparent paradox of a reduction in service (i.e., longer turnaround) with faster processing led to a search for a solution that allowed the latter without the penalty of the former. A number of schemes were developed, including the use of common disk files and two computers hooked together so that one could stoke the other's memory directly (called variously "direct-coupled systems" and "attached support processors"). These methods preserved the batch nature of a main machine and improved both efficiency and turnaround by providing faster transfer of information in and out of the main computer and by reducing the waiting time of information in the input and output streams. But the main computer efficiency was still low, and turnaround far from the instantaneous ideal. With a great deal of oversimplification, what was needed was a main computer that had a large enough memory to hold several jobs, and enough random access mass storage to hold files for all those jobs being processed, including one holding card input and printer output for each job.

Some overhead in switching from job to job is required, of course, and the operating system must be much more sophisticated if it is to keep track of what portions of memory are in use and by what jobs (especially since jobs are finishing and new ones starting all the time), and what files are in use in what positions and by what jobs. This arrangement might be referred to as a sort of simultaneous batch, but it is called "multiprogramming" (more than one program in execution at once), in fact. Although computer efficiency is greatly improved, turnaround is still far from instantaneous. Since there are only a few card readers and printers on such systems, and since it is desirable to have input and output on disk or tape whenever possible (so it can be read and written faster), a job cannot be a candidate to start until all of it is on disk, nor can its output be started on the printer until all of it has been written out to the output file (called the "output queue"). If there were many input/output devices, one could be associated with each job. Then the execution of that job could be carried out piecemeal as the input was available and as any output could be printed. In this arrangement, with typewriter-like devices for input and output, we have time sharing.

Kinds of Computing Center Applications. Early in the history of computing, and until about the mid-1960s, computers tended to be more specialized in the kinds of applications they could handle. Some of those machines were designed and built to meet the needs of commercial and industrial

COMPUTING CENTER

accounting and record keeping. These applications require fast, reliable input-output devices of large capacity but with relatively small main memory and unsophisticated arithmetic capability. Records are usually processed sequentially one at a time, as when writing paychecks for one man after another. In contrast, for engineering and scientific problem solving, machines were developed with an emphasis on their arithmetic units rather than on their input-output devices. In fact, since most scientific problems used relatively small amounts of data and produced limited amounts of output-but required extensive main memory space to store both the relatively large number of programmed instructions and all the intermediate numbers in the calculation-such machines were normally equipped with the slower (and therefore less expensive) input-output devices.

Until the mid- 1960s, the two kinds of machines were much more distinct than today's machines. Scientific machines were characterized by slow peripheral equipment, large memories, and sophisticated arithmetic instruction sets. Their arithmetic instructions often included floating-point instructions, a feature almost never required in accounting

and record keeping because answers need be computed only to the nearest cent. Commercial machines, on the other hand, were characterized by relatively limited instruction sets.

Accordingly, computing centers tended to be divided along the same lines. Programming and operating staff who understood the need of commercial users for fast reliable access to large files of information were required for the commercial or data processing centers; and technically trained programming staff who could converse with and understand the requirements of engineers and scientists were required for the scientific and university centers. In recent years, however, an increasing need by scientists and engineers for exceptional amounts of very high speed calculations has coincided with the expansion by the commercial users into much more sophisticated record keeping (involving a greater need for better arithmetic performance) and an increasing requirement by the engineers and scientists in their applications for large amounts of data, particularly input data. Thus, the distinction between what was known as scientific computing and administrative data processing has become blurred. The result is that the machines of today



Fig. 1. Computer room at Computing Center, State University of New York at Buffalo, showing parts of Control Data 6400 computer used for education and research computing, and also the UNIVAC 1106 computer used for administrative data processing.

need many fast, reliable peripheral devices with plenty of mass storage and main computers with large memory capacity and sophisticated instruction sets.

Floating-point instructions are available as an option on almost all machines, memory can be added in incremental banks of a few thousand words, and peripheral devices with a wide range of speeds and capacities can be attached. Accordingly, computing centers are becoming more diversified, although many of the specialty shops still exist, especially for the commercial users. In a large firm, for example, where an early machine installed for record keeping in the early 1960s was found to be inadequate to meet the increasing computing needs of the engineering and research departments after about the mid- 1960s, it was likely to be replaced with a machine suitable for both kinds of applications. Thus, new equipment can be configured to meet present and anticipated needs, through modular design and flexible financing, and configurations can be kept constantly in flux, changing to meet current needs. With this increasing flexibility in the equipment, many computing centers are also amply staffed to meet the needs of their users.

Physical Characteristics. Depending on the amounts and kinds of services provided, a typical computing center consists of a computer room (or machine room), a data preparation/dispatching area, a file-storage area, and offices for the personnel arranged in some logical manner. The machine room is about a third of the space (with wide variations between installations), and usually has a raised (or false) floor. This floor, usually tiled with 2 by 2 ft panels, rests on 8-14 inch pedestals above the main slab. Air conditioning and heating ducts force air under these panels, and power and control cables are also housed there. The (usually) cooled air and the cables come up through holes cut in the panels, often under the equipment modules, thus allowing each module to stand free of encumbrance by cabling or ductwork. In large systems, chilled-water piping is also housed under the panels and is hooked to the equipment through similar holes in the panels.

The machine room is usually heated and cooled, using equipment that is separated from all other areas of the center. This is done primarily because of the control nightmare that is generated by heat produced by the equipment, and also as a fire safety measure. Even in moderately cold climates, more heat is produced by the equipment in a computer room than is needed to keep the room at a comfortable temperature. Humidity also must be regu-

lated much more rigidly for reliable operation of the machine than is required for human comfort. Usually, the absolute values of temperature and humidity are not nearly as important to regulate as are their fluctuations from one side of the machine to the other. Once set in the human comfort zone, tolerances are relatively narrow, and are typically narrower for larger machines than for smaller ones. As computers are made faster and their components become smaller, the heat-dissipation problem stays about the same. The faster equipment requires narrower tolerances because of the increasing importance of timing electronic speeds in conductors at certain temperatures; the smaller components produce less heat and require shorter wires to connect them, but this does not mitigate the problem.

The data preparation/dispatching area varies from a front counter in the input-output clerk's office, in a small installation, to several rooms in a large organization for input preparation of data, keypunching, file checking and scheduling of input; and for rows, bins, and counters for dispatching output, checking updated files, and preparing the files for storage. The file-storage area is usually separated from the machine room, for security and fire protection, but is close to the equipment with which it is used (tape storage near the tape drives; disk-pack storage near disk drives, etc.).

Pricing. Pricing of computing services in the early days of computing was a relatively easy task. Each job required exclusive use of the entire main processor and all peripheral equipment for a certain length of time. Virtually every job ran in exactly the same length of time if rerun with the same data. Managers simply divided total costs plus margin for a period by the number of production hours they could expect to run the machine during that period. This calculation produced a rate in dollars per hour of running time, one that appeared fair to the user because it was reproducible and he could control the running time by varying the amount of data he submitted and by specifying (or writing) programs that were more or less efficient in processing those amounts of data. Program and system development could also be fairly easily priced in dollars per man-hour for various levels of talent. Until the advent of time sharing and multiprogramming, pricing was one aspect of computing that was a fairly conventional procedure. Many other types of services were subject to the same pricing criteria. For example, printing is much like computing in that regard. The user can control the price of his work by controlling the quality of paper used, the number-of

copies to be made, and by using or avoiding special ink colors, etc., but he cannot control the speed or width of the printing press. If a press runs at 100 copies per minute and is 10 inches wide, he may not expect a reduction in price if his copy is only $8\frac{1}{2}$ inches wide. Similarly, a computer user may not have required all available memory to run his job, but since he required exclusive use of the computer facility, the supplier's costs were not decreased.

When time sharing and multiprogramming became available, however, the user could expect that any memory not needed by his program might reasonably be sold to another customer. Hence, pricing in many centers has been subject to a great deal of controversy. The problem in multiprogramming arrangements is complicated by the fact that many measures of usage are not reproducible. For example, if a job writes a disk file, in some multiprogrammed systems the file may be written in either a large number of small blocks or a smaller number of large blocks, depending on how busy the system is (how much space is available for accumulation of large blocks, and how busy the disk storage device is). Therefore, as a result of the variability in the size of blocks that are accumulated before writing, the job can use a variable amount of computing time between writing each block. This, in turn, determines the amount of time the job spends taking up memory. The job may run in, say, 1 minute on an otherwise empty machine, and use 3 seconds of processor time during that minute (the rest being taken up by the writing out of the information on the disk file).

In a busy machine two effects are noticeable: There is less space for large blocks, so more time is spent writing a larger number of the smaller blocks; and the processor does not switch back to a particular job with any predictable regularity, since there is virtually no way to control whatever other jobs are running concurrently. The effect of both is to extend the amount of time that the job spends in main memory to perhaps 5 or 10 minutes. It still requires the same 3 seconds of processor time (because it still does exactly the steps it did before), but now they are spread over, say, 5 minutes. That is five times as long as it would take on an otherwise idle machine. In the case of pricing based on usage, this job should cost more when run on a busy machine (because it ties up memory and disk-file space longer, requires more overhead to write more blocks, and uses more overhead because it uses the processor more often, due to the reduced block sizes). In the case of pricing based on service, this job

should cost less because the turnaround time is worse.

Computing centers that are service centers (as opposed to profit centers) price their services on the basis of either service or cost. Few of the special arguments that apply to computing are not applicable to many other services within a corporate, university, or government environment. There is a paradoxical apparent simplicity but persistent overruns (with even the most competent staff) require extra attention, no matter what the pricing scheme, to insure that the processing that can be handled most profitably by the computer is what is processed. For example, if the service center that is newly installed must recover all costs from services rendered, it would have a high rate and an underutilized system. The well-heeled departments could use its services, but those not so well off (and who might be able to use the machine more profitably) could not afford the rates. The computer would be standing idle part of the time, and departments (which could use it to produce considerably more return than the incremental cost to have it running) would be doing without it. On the other hand, a computing center whose costs are fully absorbed into overhead, will be processing the unpopular chores for user departments (not necessarily the profitable ones). Since the users cannot control the costs of computing, they might as well have the computing center do whatever work the using department would like to farm out. As in any service, either pricing method has its drawbacks. The problem is further complicated by the difficulty in allocating costs in multiprogrammed and time-sharing systems. The services are a little easier to identify, but much harder to price. All things considered, computing center pricing presents about the same problem as that of any centralized service.

C. L. MEEK

COMPUTING ECONOMICS: ACQUISITION AND OPERATION

For articles on related subjects see ADMINISTRATIVE-BUSINESS APPLICATIONS; COMPUTER ACCOUNTING AND RESOURCE CONTROL; COMPUTING CENTER; GROSCH'S LAW; HARDWARE MONITOR; and PERFORMANCE MEASUREMENT AND EVALUATION.

COMPUTING ECONOMICS: ACQUISITION AND OPERATION

For articles on related **terms see BENCHMARK; FEASIBILITY STUDY; THROUGHPUT; and TURNAROUND TIME.**

Introduction of Computers in Organizations. Computers do not interface readily with the economic and managerial structure of an organization. Originally, computers were brought in to solve specific technological or organizational problems. For instance, accounting departments used them for payroll operations. Research and development departments used them for scientific calculations. Finally, they were utilized on production lines for process control. In this environment it is relatively easy to introduce a computer. A particular well-defined problem needs solution, and if the computer can provide a better solution than was previously available, then it is perfectly acceptable.

Advances in hardware technology and software systems have had a tremendous impact, not only on the numbers of computers used, but also on the manner in which they are used. Man-machine communication has become easier. Complex computer systems can be utilized by the end-users without the intervention of computer specialists. Typical examples are airline reservation systems, where the clerks interrogate the computer directly; or inventory control systems, which often bring the computer into direct contact with the salesman. In these cases computers are not introduced to solve a specific well-defined problem, but rather to improve the way an organization functions. As a result, computers affect the basic structure and principles of an organization.

Most organizations have vast amounts of data. To a large extent, data handling was previously considered to be secondary to people, money, and products in an organization, but the data in modern organizations has increased so much in quantity and complexity that it currently represents a major asset and problem. Data manipulation and information flow are critical in the management of an organization. Computers play a key role in storing, retrieving, and manipulating pertinent data, and therefore they are essential in the operation of the organization.

Computers should be introduced with special care, since they can considerably alter the information flow and decision-making process. Feasibility and systems studies are required before even deciding to install a computer, and these studies should consider first the most important problems of the company. Computers should not be used to implement current policies and organization. A new

structure of the company is sometimes needed, not to accommodate the peculiarities of computers, but rather to increase their effectiveness and to help achieve the goals of the organization. In fact, the introduction of computers is used sometimes as an excuse to effect far-reaching changes in the organization of a company.

Acquisition. After the decision is reached to introduce computers, there are many alternatives to consider. First, there is the choice between using a service bureau or obtaining one's own equipment. If the organization acquires its own computer, then there is a choice of leasing, renting, or buying the system. The organization can operate the computer itself, or it can have an independent company manage the installation. It is very difficult to decide which option best suits the needs of the organization. Some considerations are economic (Sharpe, 1969). In addition, there are many related factors such as level of support, manufacturer's reputation, local dealership, and available software.

The hardware must be evaluated in relation to the organization's needs. A processor cannot be chosen on its own merits, but must be selected for its compatibility and efficiency in the whole computer system with all attached peripherals. The processor speed can be estimated using instruction mixes, kernel programs, benchmarks, and synthetic programs (Sharpe, 1969). However, most of the expense in a standard installation is in the peripheral devices. Their performance with respect to the system must be established. There is usually a choice of acquiring peripheral devices and storage modules from the CPU manufacturer or from independent manufacturers. Site preparation (e.g., buildings) should not be overlooked; it can be as expensive as the computer hardware.

An organization acquires a computer to provide some services in a cost-effective manner. The hardware does not give these services directly. Most of the services are implemented by complicated software, such as operating systems, data base management systems, and applications systems. Hence, the organization should be extremely interested in the software that the system provides. Very often, computer manufacturers provide software free, or for a standard fee. In addition, many independent software companies sell their own software separately. The user may choose to develop his own software, but such a decision should be made carefully, since software is very expensive to build and to maintain.

COMPUTING ECONOMICS: ACQUISITION AND OPERATION

The acquisition problem is getting increasingly difficult. It is not a question of choosing a particular machine, nor is it solely a question of buying the fastest machine per dollar. Rather, it is to provide a combined hardware and software system that best suits the organization because it provides the needed services in a cost-effective manner. Even after the equipment is installed, new requirements for additional equipment and special software continually arise. In a typical installation the budget is allocated as follows: 25% hardware, 20% software, 25% personnel, 10% supplies, and 10% site maintenance, with a 10% contingency allowance for constantly rising software and personnel costs.

The investment in a computer installation is of such magnitude that the system should be allowed to evolve. The old approach of scrapping the old system and buying a new one is becoming less feasible with time, mainly because of software conversion costs. In addition, the cost of organizational upheaval caused by conversion can be prohibitive. For these reasons, evolution of systems is likely to prevail in the future.

Performance Monitoring. There are three main measures of system performance. First, *throughput* measures the amount of work that the system can perform. Second, *turnaround time* measures the amount of elapsed time for a job to go through the system. Third, *availability* measures the portion of each day that a system can do useful work. These measures are user-oriented in that they relate to the amount and quality of service the user obtains from the computer system. There are other internal measures of efficiency that do not concern the user. These measures are tools to help technicians increase the effectiveness of the hardware and software facilities.

The most widely used tools for performance evaluation are hardware and software monitors. Monitors are used to collect a wide variety of data concerning a system's operation. Hardware monitors are units appended to the original system, and software monitors consist of small routines that are invoked in different parts of the system to take measurements. The data collected by monitors can be analyzed to estimate the system's effectiveness.

The problem of data reduction is considerable. Monitors provide vast amounts of data that have little meaning unless they are reduced and interpreted correctly. For that purpose a deep understanding is needed of the system's functions and organization. Theoretical models can provide some insight on what to measure and how to interpret the

results of performance monitoring.

Performance monitoring provides the indicators that measure how effectively the system's resources are being used. These indicators can be used in two ways to increase the effectiveness of the system. First, the configuration of the system can be changed. That is, new equipment and/or programs are acquired and incorporated into the system to eliminate some of the problems detected by performance monitoring. Second, the load of user requests can be modified to better fit the available resources. One way to manipulate the load is to alter the price scale of services. The situation is analogous to any resource available in the economy. Economic indicators establish the state of the market. Then the rate of production is adjusted by manufacturers and the rate of consumption is controlled by pricing.

Pricing. Computers and their related facilities start out as free resources in most organizations. People are not accustomed to them and therefore need a strong incentive to use them. In universities they are often introduced as educational and research tools with large discounts. In companies they are included under general overhead expenditures.

Recently, people are becoming more accustomed to computers as man-machine communication becomes more flexible. This has led to an increased demand for computer-related resources. The demand for larger and better facilities is increasing at a faster rate than the price of hardware components is decreasing. As a result, computer systems are getting expensive, both to acquire and operate. Computer systems cannot, therefore, operate as a free resource. Their use and abuse will either force spiraling costs or require a set of arbitrary rules concerning their usage in an organization. It is better to control their use by pricing schemes.

There is no obvious scale by which computer users should be charged. They can be charged according to the basic resources they consume, such as processor time and memory space. Alternatively, they can be charged for services rendered, such as number of transactions processed. The quality of service, (e.g., fast turnaround) is also very important and may contribute to the cost. One of the main problems of any pricing scheme is balancing the work load. Users expect the same charge for the same computation. However, the amount of resources consumed during a computation may vary widely, depending on how busy the system is. For instance, a computation running during peak period will usually generate more overhead in the system

than one running during an idle period. In addition, resources have inherently more value during peak periods, since more computations are competing to acquire them. The manager of the computer center must find a pricing scheme that discourages use during peak periods, but which does not lead to inflation of the price paid for a computation.

Ideally, one can think of a computer system as an environment completely controlled by pricing, as in a free-enterprise system. For instance, the compiler buys processor time and memory, and charges for cards compiled. Such an environment is probably undesirable, not to mention that the overhead incurred will be prohibitive. There are still some computations that have to be heavily subsidized. A typical case is computer time used for instruction in universities. The computer center itself needs resources to improve and maintain the computer facility. Nevertheless, computer facilities should be treated as scarce resources. They are both valuable and costly.

REFERENCES

1969. Sharpe, W. F. *The Economics of Computers*. New York: Columbia University Press.
 1973. Gotlieb, C. C., and A. Borodin. *Some Issues in Computing*. New York: Academic Press.

D. TSICHRITZIS

CONCATENATION

For article on related subject see STRING PROCESSING LANGUAGES.

For articles on related terms see FILES; and **MACROINSTRUCTION**.

Concatenation is an operation wherein a number of conceptually related components are linked together to form a larger, organizationally similar entity.

In the context of string processing, concatenation refers specifically to the synthesis of longer character strings from shorter ones. In PL/I, for example, the string concatenation operation is indicated by a double vertical bar (||) so that if $W1 = 'CON'$, $W2 = 'CAT'$, $W3 = 'ION'$, then $W4 = W1 \parallel W2 \parallel W3$ is the title of this article. This kind of notation is generally used in higher-level languages.

Though still within the same general context of string construction, concatenation also is used to refer to a specific technique in defining macroinstructions, where a particular type of symbol may be a character string consisting of fixed and variable segments. When such a macroinstruction is used in a program, an appropriate string constant, given in the specifications, is concatenated with the fixed portion of the symbol during the macroexpansion process to form a complete syntactic component.

Within the general framework of files and file processing, concatenation refers to the operation of creating a collection of data by linking together several smaller collections. The resulting concatenated data set then can be processed as a single collection without relinquishing the individual identities of its components.

S.V. POLLACK AND T. E. STERLING

CONDITIONING

For article on related subject see LATA COMMUNICATIONS ; **General Principles**.

Conditioning is the term used to describe the improvements made in the signaling characteristics of leased telephone lines over those in the normal switched telephone network.

When a telephone connection is leased between two points, restrictions on the frequencies that can be transmitted over the switched network because of attenuation and related problems no longer apply. Since a leased line uses the *same path* continually, it is possible to put in special *equalizing filters* to insure that its attenuations and related parameters have much squarer wave characteristics than can be guaranteed on switched connections. It is this equalization that is called "conditioning." Standards are published of the characteristics the telephone company guarantees for lines to various degrees of conditioning (for which there is an extra charge). In such leased lines, it is also usual to have two pairs of connections to the nearest exchange (a so-called four-wire line) so that the full bandwidth can be used simultaneously in the two directions. Over the inter-exchange links, the conversations always go on different channels, but between the local exchange and the subscriber, the two directions of conversation share a normal switched telephone line.

CONFERENCE ON DATA SYSTEMS LANGUAGES (CODASYL)

Although the conditioning of a leased telephone line will considerably improve its characteristics, there will be time variations in the characteristics of the line. For this reason, the modems in high-performance data transmission systems will themselves add a further (variable) amount of equalization by use of adaptive digital filters. These enable (currently) switched telephone lines to be used at up to 4.8K bits per second (bps) and leased telephone lines at up to 9.6K bps. When higher speeds of data transmission are desired, it is possible to put in special lines that connect straight to the primary group in the telephone exchange and use 12 or more telephone channels. By these means, 48K Hz or even wider bandwidth can be used with appropriately high data rates of 50K bps or even higher.

P. T. KIRSTEIN

CONFERENCE ON DATA SYSTEMS LANGUAGES (CODASYL)

For articles on related subjects see DATA BASE AND DATA BASE MANAGEMENT; PROCEDURE-ORIENTED LANGUAGES; and STANDARDS.

CODASYL is a volunteer organization consisting of professional computing personnel from the computing industry and from computing-systems user organizations. It was formed in 1958 to attempt to standardize the languages used in computer programs and thus to permit such programs to be "machine independent." Initially, it was the purpose of CODASYL to choose and "standardize" a common programming language from among the numerous "common programming languages" being promulgated at that time, mostly by computer hardware suppliers. A CODASYL task force, organized in 1958 to work on the technical aspects of this objective, found it impossible to achieve acceptance of any language as a standard and equally impossible to integrate features of one language with another. Therefore, this task force published a new common language in 1960, called Cobol (Common Business Oriented Language). Initially, the only suppliers to accept Cobol were UNIVAC and RCA. The U.S. Department of Defense, one of the original contributors to the "standardization" purpose and an organization with a candidate for the common language, Aimaco (Air Material Command Optimum), thereupon made Cobol mandatory for all

suppliers of computing hardware and software who were bidding on defense procurements. This economic pressure resulted in persuading other suppliers to adopt Cobol also.

Experience gained with the initial language resulted in the publication of an improved version of Cobol in 1961. Another version, called "Cobol 6 1 Extended," was published in 1962, and additional features and enhancements have been added to the language almost every year since then. Cobol compilers were generally provided with all computer equipment from 1962 on. Cobol was adopted as an American standard by ANSI in 1968.

The CODASYL organization consists of: (1) an Executive Committee, (2) a Programming Languages Committee, and (3) a Systems Committee. The current CODASYL organization has evolved over the years from other organizational concepts that gave recognition to the separate interests of the hardware suppliers versus the users, as well as the short-range issues versus the long-range developments. As the Cobol language achieved acceptance, the disparity between the interests of the manufacturers of computers and the users of computers, which had to be recognized in the early existence of CODASYL, began to disappear, and the current organization evolved.

The Programming Languages Committee of CODASYL accepts comments and proposals for changes to Cobol from any competent source, reviews these proposals, modifies them, and publishes the resultant actions in a "periodic" Cobol *Journal of Development*, and approximately bi-annually publishes an updated version of the Cobol specifications.

Other CODASYL publications include "An Information Algebra," "Decision Tables (D-Tab)," and a "Data Base Task Group Report." The latter presents the features of various data base techniques and makes a comparison of the principal characteristics of each and the variance among them.

More recently, CODASYL has prepared specifications for a proposed concept of data base management and for language characteristics which would permit interaction with the data base management system from other programming languages such as Cobol, Fortran, and PL/I. Acceptance or finalization of both these principles will be time consuming, but when achieved they will presumably provide the same basis for standardization as the initial specifications for Cobol provided for standardization across a variety of then existing machine-oriented standard languages.

J. F. CUNNINGHAM

CONSTANTS

For articles on related subjects see **ARITHMETIC**; **COMPUTER**; **LABEL**; **NUMBERS AND NUMBER SYSTEMS**; **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.

For articles on related terms see **BOOLEAN ALGEBRA**; and **STRING**.

In computing, a *constant* is a value in a calculation which remains unchanged during the calculation. There are a variety of different types of constants discussed in this article-numerical, character, logical, location, and figurative constants. While it is fundamental to many aspects of a computer's internal operation that the basic reference to an item of information is by its location or address, it is more convenient to refer to constants by their values, since these are intrinsically meaningful in an algorithm. Consequently, all higher-level languages and their translators are structured to allow the inclusion of actual values, specified directly in the program rather than being read by the program as data or produced by explicit computations. Whatever mechanisms may be required to reconcile these syntactic facilities with the processor's basic address orientation are embedded in the language translator, beyond the operating range of most users and therefore "invisible" to them.

The spectrum of items that may be expressed as direct literal values transcends the numerical quantities traditionally associated with the idea of a "constant." Depending on the scope and orientation of a particular language, a variety of nonnumeric constants also are recognized and handled. Some of these serve as data items, while others may provide operational information for the program.

Numerical Constants. In most instances the user need give little thought to the inclusion of numerical constants in higher-level language statements. For many languages, their specification closely resembles conventional mathematical notation, with the compiler taking care of any necessary conversion to internal representation. Thus, the constants in the familiar distance formula

$$S = v_0 t + 0.5at^2$$

require no special form for specification in equivalent higher-level language statements:

(Fortran, Basic)	S = VO*T + 0.5*A*T**2
(Algol)	S = VO*T + 0.5*A*T↑2;
(PL/I)	S = VO*T + 0.5*A*T**2;
(Cobol)	COMPUTES = VO*T + 0.5*A*T**2.

A number of languages recognize an alternative representation for numerical constants. This form, similar to scientific notation, is particularly useful for expressing very large or very small values. Instead of writing long strings of zeros to establish a number's order of magnitude, the same value may be designated more concisely by showing only the significant digits in some convenient form, supplemented by an appropriate exponent value to adjust the scale. For instance, in languages like Basic, Fortran, and PL/I, the constant 0.000005 13 can be expressed alternatively as 5.13E - 06, 0.5 13E - 5, or even 5 1.3E - 7. (In Algol, the E is sometimes replaced by an apostrophe so that the exponential representation in that language would be 5.13' - 6; other notations are also in use).

PL/I accepts numerical constants specified in binary form. The use of a B immediately after the rightmost digit of a string of 1's and 0's identifies that constant as being to the base 2. Thus, the constant in the assignment statement **N = 11010.011~** represents the binary value 11010.011, (whose decimal equivalent is 26.375).

In certain areas of application, the manipulation of complex numbers plays an important role such that several languages include directly appropriate computational capabilities. Accordingly, there are provisions for defining complex variables and specifying complex numerical constants. Strictly speaking, there is no specific form reserved for complex constants per se. Rather, those languages accepting such data deal with complex numbers as combinations of real and imaginary components. In Fortran, for example, the statement **COMPLEX A, B** defines variable names A and B as being associated with complex numbers and instigates appropriate storage allocation. Now, supposing one wished to assign a value of $2.7 + 3.6i$ to A. The required statement would have the form **A = (2.7,3.6)**, thereby causing the compiler to treat the two arguments as the real and imaginary components, respectively, of complex variable A. Along the same lines, then, the statement **B = 4.4 * A - (1.0, - 0.8)** would produce the calculations $4.4(2.7 + 3.6i) - (1 - 0.8i)$, with the final result (equivalent to 10.88 + 15.049i) being placed in B. PL/I uses the specific form **BI** for imaginary constants, so that the equivalent statements would be written as follows:

CONSTANTS

```
DECLARE (A, B) COMPLEX;  
A = 2.7 + 3.6I;  
B = 4.4 * A - (1 -0.8I);
```

Character String Constants. Growing insight into computers' capabilities, coupled with developing human ability to identify and describe a widening variety of "computable" algorithms, have produced numerous important applications involving the processing of nonnumeric data. In response, special string-handling languages have been developed and more general higher-level languages have been equipped with facilities to synthesize and decompose character strings, search them in response to arbitrarily complex criteria, and perform a variety of other manipulations. An elemental aspect of these facilities is the recognition of constant values consisting of arbitrary combinations of letters, digits, and other symbols completely analogous to the treatment of numerical constants.

The syntactic approach generally followed in many languages is to bracket a character string constant by special delimiters, thereby identifying that string as a data item and distinguishing it from other strings whose usage is fundamentally different (e.g., names of variables). For example, the PL/I statement `DECLARE ESTRING CHARACTER (7);` defines a variable named `ESTRING`, associating that name with a string capable of accommodating seven characters. Then, the assignment statement `ESTRING = 'EEEEEEE';` fills the seven places with `EEEEEEE` (an E string if I ever saw one!). Note that the apostrophes are not part of the actual character string; they serve merely to define its extent. Distinction of an apostrophe that is an intrinsic part of a character string constant from one serving as a delimiter is handled by specifying a double apostrophe for each one to be shown. Thus, a constant consisting of the five characters `CAN'T` is indicated as `'CAN''T'`. The languages Cobol, Snobol, and some dialects of Fortran and Algol also use the apostrophe as a character string delimiter, while other Algol implementations recognize strings bracketed by `('` and `')`.

Logical Constants. The construction and representation of decision mechanisms in higher level languages often are facilitated by the availability of vehicles for specifying logical conditions, i.e., expressions describing situations whose outcome is either "true" or "false." A number of languages offer such facilities in terms of logical variables supported by operations that allow the synthesis of arbitrarily complex boolean combinations. When

embedded in a program, these expressions can form the basis for dynamically selecting one of two alternative actions. Accordingly, such variables are inherently bistable, limited by definition to one of two possible values. Hence, the standard repertoire of logical constants is similarly restricted.

Logical variables are accommodated in Fortran and Algol (where they are known as boolean variables). Both compilers recognize the literal strings `TRUE` and `FALSE` as part of their respective vocabularies. Thus, in each of the examples listed here, the names `V1` and `V2` are declared to be associated with true-or-false types of variables. (Some versions of Fortran recognize the symbols `.T.` and `.F.`.) Once this association has been defined, the assignment of one or the other alternative logical constant is straightforward. Conversion to the corresponding internal form used by the particular compiler is automatic so that evaluation of the expression proceeds without further intervention by the programmer.

<i>Fortran</i>	<i>Algol</i>
<code>LOGICAL V1,V2</code>	<code>boolean VI ,V2;</code>
<code>V1 = .TRUE.</code>	<code>V1: = true;</code>

PL/I's analogous facility is part of a more general structure for handling multiplicities of such variables, each of which independently may assume a value of "true" or "false." Such variables, which may be of arbitrary length, are called bit *strings*. In this context a logical variable is a string having a length of one bit and the two logical constants have the forms `'1'B` and `'0'B` representing true and false, respectively. Then, the PL/I equivalent of the statements shown above for Fortran and Algol would be

```
DECLARE (V1,V2) BIT(1);  
V1 = '1'B;
```

Location Constants. When working in a higher-level language, the programmer relinquishes direct knowledge and control of the storage locations from which the program is executed. Moreover, there is no reason to expect that those locations will be invariant from one run to the next. Yet, there are countless occasions requiring explicit references to specific statements, so that some vehicle must be provided through the language to give the programmer the opportunity to obtain unerring access to any part of the program. This is done by allowing him to attach a distinctive *label* to a statement, thereby establishing an identity that is fixed within

the context of the program and is impervious to its eventual location in storage. In this sense, such labels are constants and references to them are direct. **Fortran** and **Basic**, for example, allow the use of labels that resemble numerical constants but which are distinguishable by their contextual position. Thus, the assignment statement

```
22      X = 25.4 * Y ** 2
```

is equipped with the *label constant* 22 so that, at some other point in the procedure, one could write **GO TO 22**, thereby specifying a fixed directive to continue the processing from a particular point. Other languages differ with regard to the allowable construction of labels, but their use as constants still persists.

Extension of this idea of a location constant is seen in languages like **PL/I** and **Algol**, where label constants are constructed from alphanumeric characters, as are variable names. These languages allow components that act as location variables in that they may assume different "values" at various points during the execution of a program. For instance, the **PL/I** statement **DECLARE PLACE LABEL;** establishes a variable named **PLACE** whose "value" at any given time designates a particular statement in the program. This value is defined, conventionally, by an ordinary assignment statement. Thus, **PLACE = THERE;** defines a specific spot in the program, represented by the location constant **THERE**. Then, a reference to **PLACE**, e.g., **GO TO PLACE;** implies a reference to the statement with label **THERE**. Later on, it is possible to set **PLACE** to a different location constant, with the result that the same reference to **PLACE** now will lead to some other particular point in the program.

Figurative Constants. There are special types of constants that represent fixed and unambiguous values but which are unlike the other types discussed heretofore in that they are not designated by their literal values. Because their usage is particularly common, these fixed quantities are indicated by permanent parts of the language vocabulary. Such constants, known as "figurative constants," are provided as conveniences to enhance the "naturalness" of higher-level language statements. **Cobol** includes several such constants. For instance, a given variable may be set to a value of zero by any of the following statements:

```
MOVE ZERO TO X.
MOVE ZEROS TO X.
MOVE ZEROES TO X.
```

in which **ZERO**, **ZEROS**, and **ZEROES** are figurative constants. Similarly, the character string **B** may be filled with blanks by either of the **Cobol** statements **MOVE BLANKS TO B**, and **MOVE SPACES TO B**.

The **Snobol** language has another type of figurative constant to provide apparent visibility for nothing. This constant, **NULL**, refers to the absence of any numbers, letters, or any other items. For example, a character string named **Y** can be emptied (depleted and, in an operational sense, reduced to a length of zero characters) by the statement **Y = NULL**. An alternative form to accomplish the same thing, namely, **Y =** is not much more mysterious in the overall context of the language. However, there are many more complex situations in which the explicit appearance of something for nothing is very helpful in clarifying the nature and intent of intricate string-processing constructions.

REFERENCE

1969. Sammet, J. *Programming Languages: History and Fundamentals*. Englewood Cliffs, N.J.: Prentice-Hall. (See especially Section III.4.)

S. V. POLLACK

CONTENTION

For articles on related subjects see **COMMUNICATIONS AND COMPUTERS**; **LOCKOUT**; **MULTIPROCESSING**; and **MULTIPROGRAMMING**.

For article on related term see **BUFFER**.

Originally, the term "contention" was used to describe a communication system where the terminals, or lines, were competing for a circuit and the first one to find it free obtained it. This concept can be generalized to the case of multiple users (jobs, tasks, processes) competing for sharable resources (processors, channels, devices). For example, in a multiprogramming system, two jobs may simultaneously require the use of tape drives, thus possibly exceeding the capacity of the installation. This overflow situation would lead to contention delays, since one job would have to be put temporarily in a waiting state. Another example can be found in the case of a multiprocessing system where a process can be split into several tasks and the number of tasks

CONTROL APPLICATIONS

ready to be processed in parallel is larger than the number of available processors.

Contention is solved by using priority schemes; the simplest one is a first-come-first-serve strategy. However, all processes contending for a resource must be remembered so that they will, in turn, be able to use it. This implies the presence of queues, or buffers, associated with each sharable resource.

J-L. BAER

CONTROL APPLICATIONS

For articles on related subjects see **ANALOG COMPUTERS**; **DIGITAL TO ANALOG CONVERTERS**; **MINICOMPUTERS**; **REAL TIME APPLICATIONS**; and **SPECIAL PURPOSE COMPUTERS**.

For articles on related terms see **ALGORITHM**; **ASSOCIATIVE MEMORY**; **BLOCK DIAGRAM**; and **MODELS**.

The history of automatic control traces back thousands of years to the water clock used by the Egyptians in the third century B.C. This early clock consisted of a jar from which water escaped at a controlled rate. The level of the water in the jar indicated the time of day. Only since the Industrial Revolution, however, has there been a conscious effort by man to analyze elements of his environment and to control them automatically.

By 1950 there was a unified body of knowledge called "feedback control theory," which could be used to analyze and control the performance of complex devices and processes. The development at that time of the digital computer, capable of making hundreds of thousands of computations and logical decisions in 1 second, introduced a powerful new tool for solving conventional control problems. Furthermore, it uncovered many new theoretical and applied research problems for control theorists and engineers.

Although the full potential of the digital computer as a control tool has not yet been realized, its use in a wide variety of control applications is growing rapidly.

In the remainder of this article, we will explain what is meant by the term "automatic control" and discuss the role of the digital computer in the solution of control problems.

As the term "control" implies, in every control problem there is a system (which may be a device, a process, or some phenomenon) with associated output or **controlled** variables that are to be forced to satisfy certain constraints. Typically, output variables cannot be adjusted freely; instead, they depend in an indirect way on another set of variables that can be adjusted. The variables in this second set are termed "input" or "manipulated" variables.

The nature of the dependence of the output variables on the input variables is **very** often complex and cannot be determined precisely. However, the relationship that exists between the two sets of variables can often be approximated by a set of differential (or difference) equations, which is referred to as the "mathematical model" of the "system." We will represent this model, with its input and output variables by a block diagram, illustrated in Fig. 1.

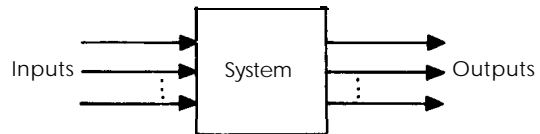


Fig. 1. Block diagram of model of system.

It may be that a given system performs very well as is. On the other hand, it may display unfavorable characteristics such as instability, sensitivity to parameter changes, slowness of response, or inefficient operation. Control theory embraces the problems of developing suitable mathematical models, determining system characteristics from the mathematical models and altering those characteristics when they are unacceptable.

Consider, as an example of a practical control problem, a lawn mower with a small internal combustion engine and a speed-control lever connected to a butterfly valve in the carburetor. The butterfly valve regulates the flow of fuel into the cylinder.

In control terms (Fig. 2), the output of this system is the engine speed S_e . With a constant load applied to the engine, the engine speed is determined

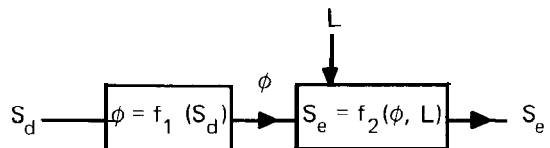


Fig. 2. Lawn mower model.

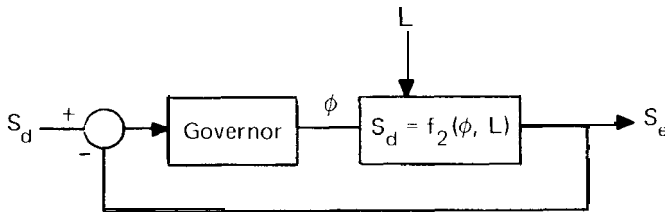


Fig. 3. Lawn mower model with feedback control.

by the angle ϕ of the butterfly valve in the carburetor. The angle ϕ is a function of the desired speed S_d , which is considered to be the system input and is set by the operator with the speed-control lever.

The problem with this system is that the speed of the engine (S_e) in actual operation is determined by the angle ϕ and the load L on the engine. With ϕ held constant, an increase in the load will cause a decrease in engine speed. Consequently, if one attempts to use this machine to mow a lawn that is uneven and has thick patches of grass, he will find that the speed-control lever must be frequently adjusted to keep the engine running at a constant speed.

An intuitively simple solution to this problem (Fig. 3) utilizes *feedback control* to make the system less sensitive to load changes. Here the angle ϕ is forced mechanically to be a function of S_d , the desired engine speed, and S_e , the actual engine speed. The control scheme works in the following manner: With the use of springs, levers, and balanced weights, the engine speed S_e is continuously measured and used to change the angle ϕ in proportion to the engine speed error $S_d - S_e$. As a consequence, an increase in the desired speed (caused by the operator) or a decrease in actual speed (caused by an increase in the load on the engine) results in an increase in fuel flow to the cylinder. This action tends to reduce the engine-speed error to zero. The mechanism that measures engine speed and makes the angle correction is called a "governor."

Many other examples of automatic control problems, like the one discussed above, have simple solutions that can be implemented at a low cost. On the other hand, there are many complex automatic control problems requiring sophisticated control schemes that cannot be easily implemented with mechanical devices or with simple electronic feedback circuits. It is in the solution of these problems that the digital computer is most effective.

We will now discuss several areas of control where the computer is being utilized. In each of these

areas the motivation or justification for computer control is one or more of the following: More flexible control, improved product quality, greater safety, greater precision, More efficient operation, or a savings in manpower or equipment,

Process Control. Process control is the term given to the automatic control of industrial processes such as paper machines, blast furnaces, chemical plants, electric power generating stations, sewage treatment plants, and the like. The typical digital process control system may be expected to achieve either one or a combination of the following objectives :

1. Make the necessary adjustments so that the process recovers from upsets and disturbances in the most expeditious manner.
2. Determine the level of process operation that achieves the maximum economic return on investment.

The second objective is in reality an optimization problem, but in order to implement successfully the optimum conditions, a necessary prerequisite is an effective plant control system.

To illustrate these two types of objectives, suppose we consider the simple hot-water heater in Fig. 4. As shown, the control system admits steam at

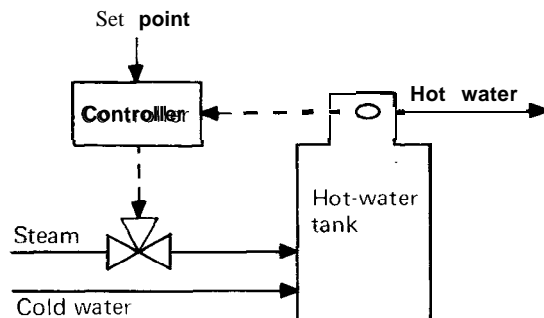


Fig. 4. Hot-water heater.

CONTROL APPLICATIONS

the proper rate so as to maintain the outlet water temperature at some desired value, usually called the "set point." The outlet water temperature is measured by the *sensor*, whose output is called the "feedback variable"; this outlet value is compared with the desired value (set point) to generate the error signal. In effect, the objective of the controller is to drive the error signal to zero. In so doing, the controller generates a signal, referred to as the "manipulated variable," which in this example determines the opening of the steam valve and therefore the amount of steam flow.

The relationships between the elements in the system are best illustrated by the block diagram in Fig. 5. The process consists of the hot-water tank, which is subject to two inputs. One input is the steam flow. The second input is the water flow through the tank, referred to as an "upset" or "disturbance," since changes in this flow affect the outlet temperature. These two inputs determine the flow of water from the tank and the temperature of the water; the latter is to be regulated by the control system, and is called the "controlled variable." Its value is measured by the sensor and fed back to the comparator to generate the error signal. On receiving this signal, the control algorithm generates a signal to the actuator, which in this case is a steam valve. The valve then adjusts the flow of steam into the vessel. The elements in Fig. 5 are arranged in a loop, and the system is frequently referred to as a "control loop."

In all loops in older industrial plants and in many loops in newer plants, the comparator and control algorithm are implemented in a hardware unit known as a "controller." In effect, this unit is a special-purpose analog computer, with the computing elements being either pneumatic, electronic, or occasionally hydraulic. This unit continuously monitors the feedback variable and set point, and generates adjustments to the steam valve in a continuous fashion.

With the appearance of digital computers, the possibility of replacing the analog (conventional)

controllers by a digital control system was investigated; the first such plant installation appeared in the late 1950s (for a modern installation, see Fig. 6). To be economically feasible, a single general-purpose digital computer had to assume the responsibility of conventional controllers. Therefore, the computer's time was shared among many control loops. In effect, the computer services the loops in turn, i.e., computes the error signal, performs the algorithm computations, and proceeds to the next loop. The time between *servicings* for a given loop is referred to as the "sampling time," which is not necessarily the same for all loops. The use of digital computers in this fashion is called "direct digital control" (DDC).

In digital control, the output (the signal to the steam valve in Fig. 4) is usually computed from the error signal (e_n is the error at sampling time n) by an equation (the control algorithm) containing three terms :

1. A proportional term, $K_c e_n$, where K_c is a constant, the "proportional gain."
2. A summation or integral term, $(T/T_i) \sum_{i=0}^n e_i$, where T_i is a constant, the "reset time," and T is the sampling time.
3. A difference or derivative term, $T_d(e_n - e_{n-1})/T$, where T_d is a constant, the derivative time.

The output of the controller is simply the sum of the values of these three terms. The proper values of K_c , T_i , and T_d depend upon the process being controlled, and the adjustment of these constants to their proper values is a procedure known as "tuning." Although other control algorithms have been proposed, over 90% of the control loops in industry use this three-term, or proportional integral derivative (PID), algorithm. In some control loops the derivative term is omitted, leaving the two-term, or proportional integral (PI) algorithm.

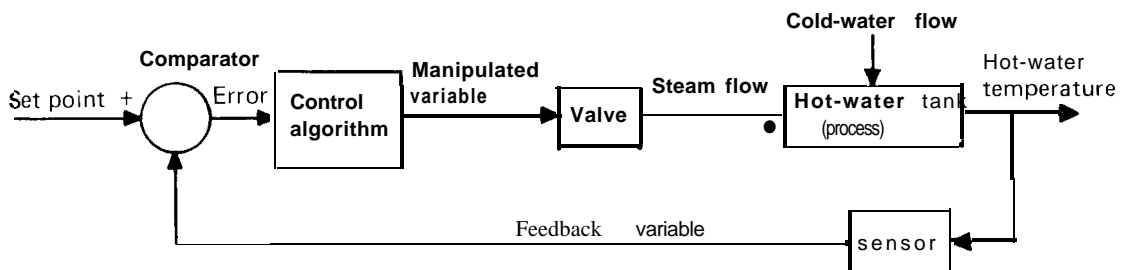


Fig. 5. Control loop.

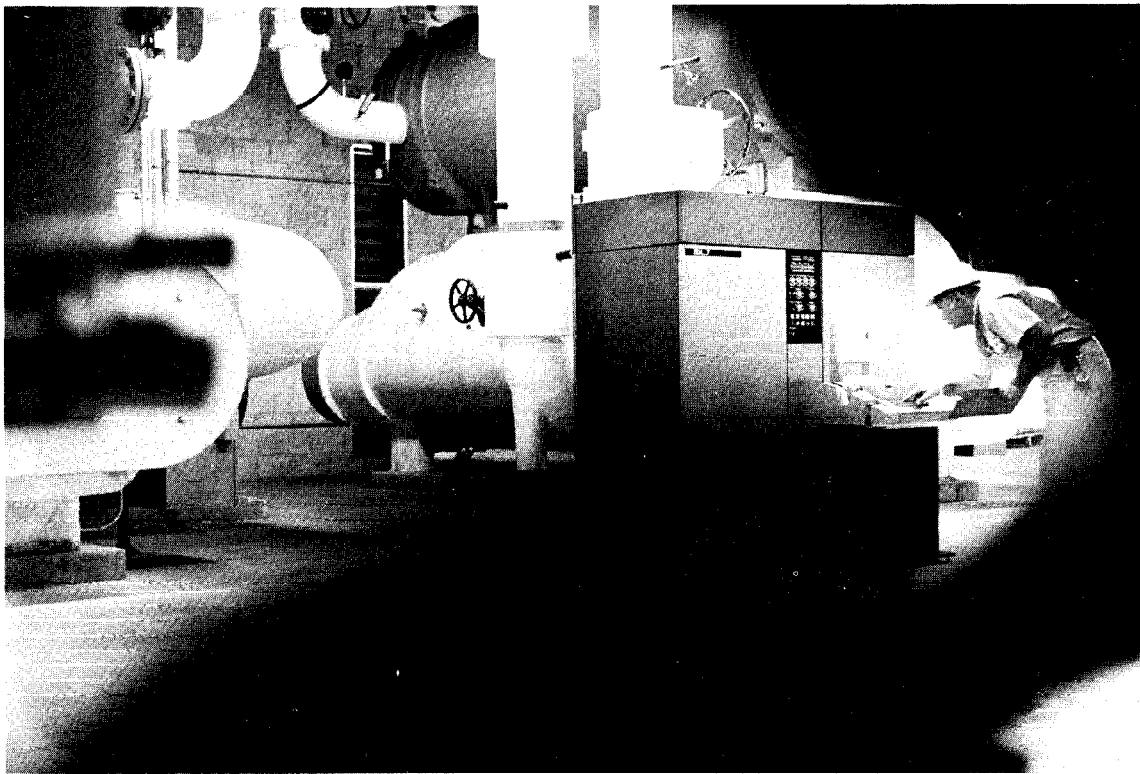


Fig. 6. An IBM System/7 running in a process control environment.

The control system in Fig. 4 must perform satisfactorily for two somewhat different control problems. To illustrate, suppose the system is functioning in a large hotel. Each time a guest opens or closes the hot-water faucet in his room, he changes the flow of water through the tank. Suppose he also opens the cold-water valve. This effectively admits more cold water into the system, which tends to lower the outlet water temperature. However, the control system detects this change, and admits more steam to compensate. In this case, the control system is trying to maintain the controlled variable at the set point in face of upsets. This is the *regulator problem*.

To illustrate the second problem, suppose the hotel manager is unusually responsive to criticism from the guests so that each time a guest complains about the water being either too hot or too cold, he either raises or lowers the set point to the controller. If he raises the set point, the control system must respond by increasing the flow of steam so that the outlet temperature will increase. This control is the *servo problem*. The three-term algorithm previously described works well for both types of problems.

The current trend is to use minicomputers in

most industrial direct digital-control systems. However, particularly in the chemical and petrochemical industry, direct digital control of the total plant has frequently not proved to be economically attractive. The conventional controller is relatively inexpensive compared to even small digital systems, and performs the three-term algorithm computation quite well. However, the digital computer is able to perform other services (e.g., feedback variable compensation, such as correcting flow measurements for temperature effects). In these cases, the flexibility and computational capabilities of the digital computer make it more attractive than conventional controllers. In many plants, those loops requiring this capability are implemented digitally and the others are implemented conventionally.

To illustrate the optimizing control problem mentioned previously, suppose we have a hot-water heater in which the heat source may be either steam, electricity, gas, or combinations of the three, as shown in Fig. 7. In this situation, the control system must choose which heat system to use. The choice should depend on which source can produce the hot water at the least cost. Any one of the three sources

CONTROL APPLICATIONS

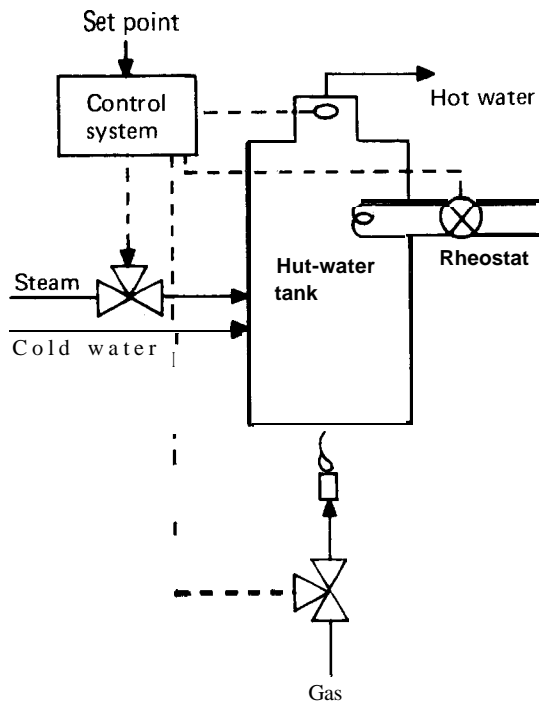


Fig. 7. Optimizing control problem.

or a combination of all may be chosen. A combination may be necessary if, for example, a limit (constraint) is imposed on the amount of heat available from one source. Changing demands from night to day, changing availabilities of energy, changing cost of energy (electricity is often cheaper for industrial use during the hours after midnight), and similar factors require that the optimum requirements be recomputed on a frequent basis.

In many industrial systems, the output of the optimizing control system in Fig. 7 would not be set directly by the valves but would be determined by the values of the flows. In case of the steam flow, for example, a separate control loop must be installed to sense the volume or pressure of the steam flow and adjust the valve automatically to produce the flow specified by the optimizing control system. In this case, the optimizing control system is frequently referred to as a "supervisory control system."

Traffic Control. The control of traffic, both in airways and on roadways, is becoming a critical problem. Projection of traffic volume into future years indicates that the present means of control is not a satisfactory long-term solution. Although air and ground traffic share the common goal of in-

creasing the volume of traffic flow, the two problems have different natures.

Control of air traffic in and out of airports is now, as it has been in the past, the responsibility of air traffic controllers; these people from the air traffic control center monitor traffic data and give takeoff and landing instructions to aircraft pilots. The volume of controlled traffic through an airport is limited to the maximum number of aircraft that air traffic controllers can safely handle. This maximum number depends to a great degree on the form and extent of information available to the controller.

In the early years of air traffic control, the standard "real time" information available to the controller consisted of a flight schedule, aircraft position and altitude communicated from each pilot by radio, and an overall view of close-range traffic movement shown on a radar screen.

As the volume of traffic increased over the years, the need for more sophisticated systems for gathering, processing, and displaying flight information grew accordingly. The past decade has seen a major effort to utilize the computer and other electronic devices to develop a data handling system capable of meeting the requirements of air traffic control today and in years to come, although much needs yet to be done in this direction.

In England the first stage of a national air traffic control program, named *Mediator*, has been introduced (Editor, *Wireless World*, 1971). This system incorporates primary radar, together with a secondary radar facility that works with those aircraft equipped with transponders. These radar signals are processed by computer and distributed to graphics display units, which provide air traffic controllers with aircraft identification and route codes superimposed on the primary radar display. Altitude display is also provided for those aircraft equipped with altimeter telemetry.

The second stage of *Mediator* calls for the inclusion of a flight-plan processing system that will notify controllers of any aircraft which deviates from its flight plan or is flying a collision course.

Although systems like *Mediator* are effective, it is estimated that a maximum number of 300 aircraft can be tracked and identified using a conventional processor which must retrieve a word from memory before it can be used in logical operations. However, the number of aircraft expected to be under terminal control of large airports by the year 1980 is 1,500. Now in the experimental stage are two computers with *associative* processors (Editor, *Electronics*, 1970; Eddey, 1970), which have logic circuits at every word, or even at every bit location in memory. An

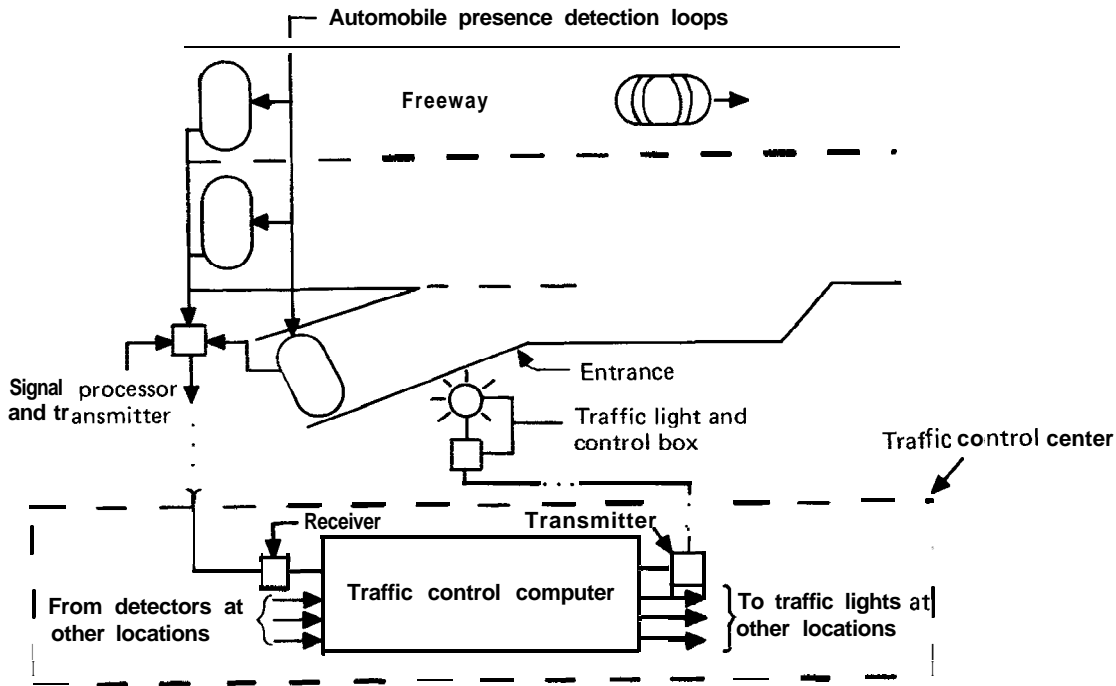


Fig. 8. Freeway traffic control.

exact match search of all bits in memory can be performed by these processors in 100 to 300 ns. With such speeds, it appears that these processors will be capable of monitoring a large number of aircraft and identifying those that are traveling on collision courses.

Ideally, one would like to have a computer monitor traffic data and compute, for each aircraft, an optimum route to be traveled. However, because so many lives depend on air traffic control, highly reliable equipment together with backup systems and fail-safe procedures must be developed before closed-loop computer control can be employed.

Control of traffic on the freeways and surface roads of large cities is another problem that is receiving increased attention. Unlike air traffic control, the traffic routes are fixed, and the only means of control are traffic lights and possibly roadside messages to automobile drivers.

In London (Holland, 1969), a computerized traffic control program has been in operation for some time and has increased traffic volume considerably. Detectors in the streets provide traffic flow data, which is processed by a central computer to determine optimum sequencing of traffic lights.

In Los Angeles, California, presence detectors have been installed on freeways to gather data for

the development of mathematical models of freeway traffic as well as algorithms for parameter estimation, incidence detection, and flow optimization during times of peak loads and abnormal conditions, as illustrated in Fig. 8. After models and control schemes have been worked out, the loop will be closed and the computer will implement control schemes by regulating the flow of traffic at freeway entrance ramps.

Defense Applications. The early development of the electronic digital computer was within a military framework. Computer control is still used extensively in the armed services today but, compared to applications in industry and other commercial enterprises, the justification for its use is less dependent on economic considerations.

Although the details of most defense systems are classified, it is obviously true that the firing of many offensive and defensive weapons is performed under the control (either direct or supervisory) of a computer. Many applications that were initially developed within the defense department are now being used commercially. Automatic flight control with the use of inertial navigation, for example, was first developed to control the flight of ballistic missiles and is now being used for communications

CONTROL APPLICATIONS

and weather satellite deployment, manned space flight, and control of commercial aircraft. This topic will be discussed in more detail in the next section of this article.

Applications in Air and Sea Transportation. Computers are now being used for control purposes on board air and sea vessels. Although computer control on board merchant ships has lagged its use on shore, some vessels (such as Queen Elizabeth 2; see Phillips, 1971) utilize computers for direct as well as supervisory control.

Feedback control is used aboard the Queen Elizabeth 2 (*QE2*) to regulate scoop pumps that provide circulating water to the condensers in order to increase the efficiency of the engine and thereby decrease fuel consumption. The recommended relationship (provided by the turbine designer) between condenser vacuum and engine shaft-horsepower is stored in the computer. Computations are made periodically to determine the shaft horsepower of the engine and the corresponding recommended condenser vacuum. The actual vacuum is compared with the recommended condenser vacuum, and the change in vane angle of the scoop pump necessary to give the recommended vacuum is computed. Finally, the computer sends a signal to an actuator to make the correct change in vane angle.

The computer in the *QE2* is also utilized for supervisory control to calculate the recommended power output, engine speed, and shaft speed of the ship over the route to be traveled. To accomplish this, the desired route is divided into as many as 20 sections. Over each section, wind speed and direction as well as ship's course are assumed constant. The calculations are based on weather reports and forecasts, the ship's load, and scheduled arrival time.

Plans are being made to have the computer control on a daily basis the generation of fresh water according to predicted usage. Scheduled stops would be considered to insure that fresh-water tanks are full when the ship stops and the evaporators are shut down.

In addition to its use on the ground for air traffic control, as discussed previously, the digital computer is being used on board commercial aircraft for control purposes, such as flight control and control of power distribution.

The Boeing 747 is equipped with an inertial navigation system, a compact version of systems used to control the flight of missiles and space vehicles. Gyroscopes are used to hold a platform in a fixed, level position with respect to true north. Two accelerometers placed on this platform continuously

and accurately measure acceleration along a north-south axis and along an east-west axis. The output of each of these accelerometers is integrated with respect to time to determine aircraft velocity with respect to the two axes. Finally, the velocity along each axis is integrated with respect to time to determine the north-south and east-west components of the plane's position with respect to the point of departure. The integration is performed by a digital computer.

With the coordinates of the destination point stored in memory, the digital computer can easily compute correct headings and, with some additional information provided, calculate pilot control settings. Although closed-loop control is not presently being used, evidence indicates that computer control would be more accurate than pilot control for maintaining altitude during turns and for maintaining a correct heading during flight.

Computer control of electric power distribution in aircraft and other vehicles promises to reduce system complexity and cost while providing better performance. A prototype model of such a system has been constructed (Editor, *Product Engineering*, 1970). In this system (Fig. 9), a central digital computer controls the action of remote hybrid electronic power controllers that are located at load centers and which perform the switching. Load control and sequencing is under command of the central computer, which can be programmed to perform automatic self check-out, startup, and shutdown sequencing, and load shedding on a priority basis when generating capacity is reduced.

Numerical Control. In the manufacture of parts for such products as appliances, automobiles, airplanes, and the like, fabrication of the part is effected by such devices as milling and routing machines, rotary tables, lathe grinders, boring machines, flame cutters, drilling machines, and benders. When using any of these machines, some type of control mechanism is necessary to insure that the finished part meets specifications.

In the early manufacturing plants, such control was performed by a human operator. This mode of control was relatively expensive, and the consistency with which the resulting part met specifications was relatively low, especially for parts requiring extensive milling. With the development of the automotive industry, machines were designed with preset tool guides, automatic part holding and locating features, automatic feed systems, and the like. These machines, referred to as "fixed" or "Detroit automation," were designed to produce a specific part.

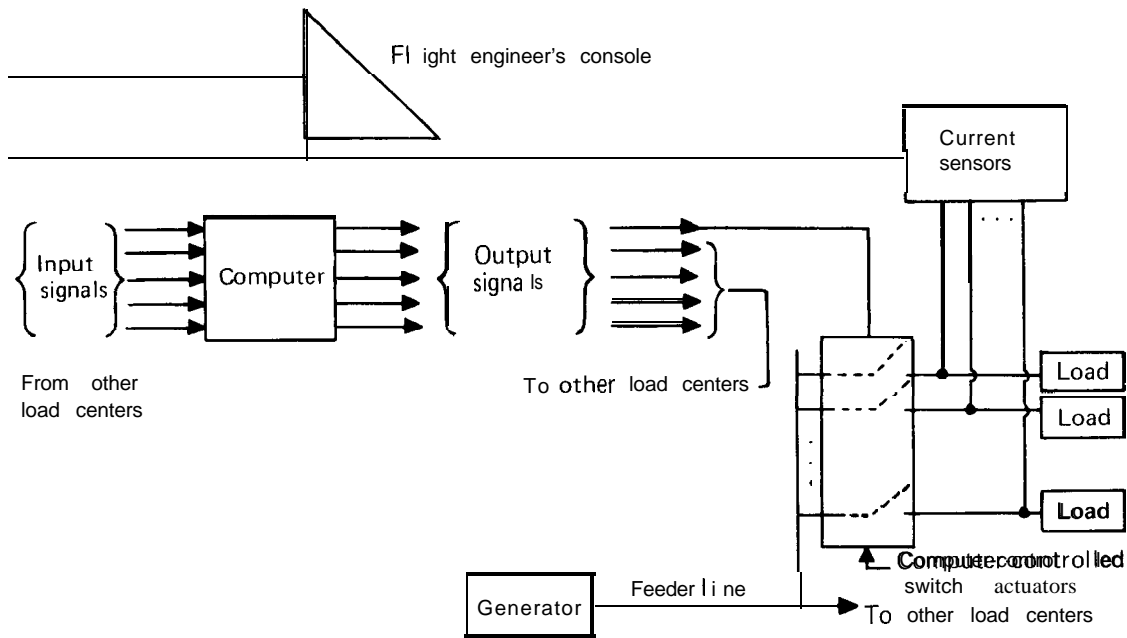


Fig. 9. Control of aircraft power distribution.

Although part changes frequently required major mill modifications, this was no great obstacle because of the large quantities of each part produced in the automotive industry.

Many major industries, such as the aircraft industry, must fabricate parts in considerably smaller quantities. In these applications the mill should have the flexibility to produce a variety of parts with a relatively short changeover time. The milling machines generally have capabilities such as positioning the part in either the two- or three-dimensional coordinate system, rotating the part as in a lathe, positioning a cutter as in a drill, and the like. When the instructions that specify the positioning, movement, and other machine operations are in the form of numbers, the machine is a candidate for numerical control.

The control of the machines by numerical control can be classified in two types: point-to-point control and contouring control. In point-to-point control, the route by which the tool moves from point A to point B is immaterial except that, among other considerations, the movement must be sufficiently precise so that the tool does not strike the part. Drilling multiple holes in a part is an example of this type of control. In contouring control, milling is usually done along a path, which must be controlled at every point. This type of control is the more demanding.

Basically, an order for a part generally includes a drawing of the part and some additional explanatory information. From this data, the plant must produce the part. The steps are frequently as follows:

1. The dimensions and other pertinent data of the part are punched onto cards.
2. These cards constitute the input to computer programs that provide a generalized milling and positioning format.
3. The resulting data is then entered into a post-processor program, specific to the machine being used, to produce the numerical control instructions, usually in the form of a punched paper tape.
4. This paper tape is mounted on a paper-tape reader at the machine to provide direct instructions to the machine.

To convert from manufacturing one part to the next, the mill operator need change only the paper tape and make the usually minor modifications on the mill. To facilitate this **changeover**, many machines have selectable tools, e.g., multiple drill heads mounted on a turret.

REFERENCES

1969. Holland, Ted. "London's Computerized Traffic," *Engineering*, Vol. 207 (March 7), pp. 374-375.

CONTROL DATA CORPORATION

1970. Eddey, E. E. "Ground Based Aircraft Collision Avoidance Using an Associative Processor," *IEEE Proc. National Aerospace Electron Conference* (May 18-20), pp. 302-306.
1970. Editor, *Electronics*. "Air Traffic Control by Associative Processors," *Electronics*, Vol. 43 (July 6), p. 40.
1970. Editor, *Product Engineering*. "Plane's Electric System Is Switched by Computer," *Product Engineering*, Vol. 41 (December 7), p. 61.
1971. Phillips, H. "Computer Systems for Merchant Ships," *Electronics and Power*, Vol. 17 (January), pp. 35-39.
1971. Editor, *Wireless World*. "Progress in Traffic Control," *Wireless World*, Vol. 77 (April) p. 77.

C. L. SMITH AND B. MOORE

CONTROL DATA CORPORATION.

See MANUFACTURERS, COMPUTER.

CONTROL DATA CORPORATION 6000 SERIES

For articles on related subjects see **MANUFACTURERS**, **COMPUTER**; and **SUPERCOMPUTERS**.

For articles on related terms see **CENTRAL PROCESSING UNIT**; **INTERLEAVING**; **MULTI-PROCESSING**; **MULTIPROGRAMMING**; **PARITY**; **REGISTER**; and **STACK**.

The Control Data 6000 series consists of the CDC 6600 and its variants. The 6600 was designed to provide high-speed arithmetic capability for large scientific applications, particularly solutions of very large systems of linear and nonlinear equations. The 6600 was both innovative and successful. The innovation lay in an architecture that employed an unprecedented degree of concurrency. The success was such that from the time the first 6600 was delivered in 1964 until about 1970, the CDC 6000 series dominated the large-scale scientific market. These machines were deservedly called the "biggest and most powerful in the world."

A CDC 6600 has been described (Bell and Newell, 1971) as a "network computer"; it consists of a central memory (CM), a central processor (CP), a set of peripheral processors (PPs), and peripheral

equipment. Extended core storage is available. Each PP has its own memory (Fig. 1), and although some arithmetic circuitry is shared among the PPs, a 6600 with 10 PPs is programmed as an 11-way multi-processor. Since I/O and operating systems functions may be performed concurrently with user programs, it is estimated that only 2% of central processor cycles are consumed by operating-systems overhead.

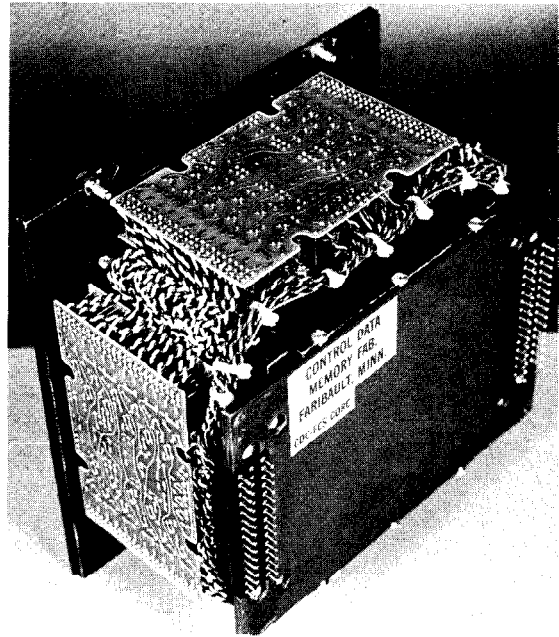


Fig. 1. The CDC 6000 memory unit.

The basis of the 6600 central processor architecture is "functional parallelism" (Thornton, 1970). It is achieved by the use of several functional units plus support facilities. In a typical central processor program, at least two or three functional units will be in operation simultaneously. Instructions from memory are first loaded automatically into a high-speed scratchpad memory (the "instruction stack"). Instructions are saved in the stack so that under some circumstances a program loop can be held, and the program can loop within the stack at high speed without requiring central memory references for instructions. The unit and register reservation control, or "Scoreboard," schedules instructions and the functional units, taking an instruction from the stack whenever a unit is free.

The 6600 is able to add two 18-bit quantities in 300 ns, to add two floating-point numbers in 400 ns with 14-place precision, and to multiply two float-

q-point numbers in 1 μ s. Since these operations use different functional units, they may be performed **simultaneously**. It is estimated (Grishman, 1971) that the processing unit of the 6600 typically executes **about** three million instructions per second. There are no interrupts in the usual sense.

The maximum memory of a 6600 consists of 13 1,072 sixty-bit words. The memory is divided into 32 *banks* with *low-bit interleaving* so that consecutive addresses are assigned to consecutive banks and references to consecutive addresses may be overlapped. Protection is by base-and-bounds registers not addressable by the programmer, and **relocatability** is such that programs may be freely moved about in memory in order to solve the storage-fragmentation problem of multiprogramming.

The other members of the CDC 6000 series are the 6400, 6500, and 6700. The 6400 is a slower and simpler version of the 6600, with a "unified arithmetic" unit instead of the ten functional units, and no instruction stack. The 6500 and the 6700 are dual processors, the first consisting of two 6400 processors and the second of one 6400 and one 6600.

In spite of the undoubted success of the 6000 architecture, some criticisms can be made. Floating-point round is performed *before* rather than after normalization, in contravention of the principles of numerical analysis. The subroutine-jump instruction stores into memory, thus making reentrant code difficult to implement. Fixed-point overflow is not detected. One's-complement arithmetic presents some problems, particularly in the handling of the two representations for zero. There is no parity checking on memory, making intermittent memory failures difficult to diagnose.



Fig. 2. A Control Data Cyber 70 system, a modern version of a 6000 series system.

In 1969 the Control Data Corporation introduced the 7600 system, which is about five or six times faster than the 6600. The 7600 is similar in organization to the 6000 series, but it uses faster circuitry, a faster memory, and a large backing store (magnetic core) in addition to the usual core memory directly addressable by the processor. Memory parity checking has been introduced. In the 1970s, the 6000 series and the 7600 were reintroduced as the Cyber 70 series (Fig. 2). The organization is again basically the same, the chief enhancement being a **COMPARE AND MOVE** instruction, which allows for rapid character manipulation. This addition makes the Cyber 70 line attractive, not only to the scientific community, but to the business community as well.

REFERENCES

- 1970. Thornton, J. E. *Design of a Computer: The Control Data 6000*. Glenview, Ill.: Scott, Foresman.
- 1971. Bell, C. G., and A. Newell. *Computer Structures: Readings and Examples*. New York: McGraw-Hill.
- 1971. Grishman, R. *Assembly Language Programming for the Control Data 6000 Series*. New York: Algorithmics Press.

A. H. WERKHEISER

CONTROL POINT

For article on related subject see **MICRO-PROGRAMMING**.

For article on related term see **REGISTER**.

The hardware locations at which the output of the instruction decoder of the processor activates the input to and output from specific registers as well as the operational resources of the system (adders, shifters, etc.) are called the "control points." The control points basically determine intercycle register-to-register communications. For each register in the processor there are a fixed number of other registers to which data may be transmitted in one cycle. For each such possibility, a separate **AND** circuit is placed on the output of each bit of the source register, with the entry into the destination register being collected from all possible sources by an **OR** circuit.

CONTROLLED VARIABLE

Data may be communicated from one register to another directly or through some operational network (adder/shifter, etc.). Transmission to such a network must also be selected; the output of the network has a similar set of gates for each possible destination. In a typical system there must be control points corresponding to each bit of every register for each possible intercycle destination of that register and for each bit of each operational network that may be used in a cycle.

For example, consider a 32-bit computer with eight registers. Assume that each register can communicate with three other registers in one cycle. The number of control points required for register communication would therefore be $3 \times 8 \times 32$, or 768. In addition, assume the machine has three execution resources, each of whose 32-bit outputs can be gated to one of four registers. This would account for an additional $3 \times 4 \times 32$, or 384, control points. Of course there are additional control points for the selection of a particular function within a designated resource. This might account for 100 more control points. Thus, there is a total of somewhat over 1200 control points, which must be established for each cycle by the output of the instruction decoder (the control function). Fortunately, in most computer design situations, many of these control points are not independent. For example, one may not gate bit 7 of register 4 to another register, but one may gate the entire contents of register 7 to its destination register. Since only one line is required to control such multiple control points, the total number of outputs required can be significantly reduced. These outputs are then referred to as "independent control points." For the hypothetical system described, there might be anywhere from 50 to 200 independent control points, depending upon the variety of instructions in the vocabulary of the system.

M.J. FLYNN

CONTROLLED VARIABLE

For articles on related subjects see **ITERATION**; and **PROCEDURE-ORIENTED LANGUAGES**, Programming in,

The term "controlled variable" is used for the variable that is controlled to take on a specific set of values in an iterative structure in a programming language. The number of values in this set determines or partially determines how many times the

statements in the iteration are executed. For example, in the Algol statement

```
for i: = 1 step 2 until j, j*j while k < m, 6, v, x*y
do begin (iteration statements) end
```

i is the controlled variable. Its values—1, 3, 5, . . . as long as $i < j$, then j^2 as long as $k < m$, and then the three values 6, v , $x*y$ —are taken on successively each time the statements in the iteration are executed. Only the value j^2 may be taken on by i during more than one execution of the iteration statements; this value will be retained as long as $l < m$ (where l or m , or both, will be changed by the iteration statements). For this reason the sequence of values of i only partially determines the number of iterations. By contrast, in the Fortran structure

```
DO 212 I = 1, 13, 3
```

(iteration statements, with last one labeled 212),

The variable I takes on the values 1, 4, 7, 10, 13, once only, each time during the execution of the iteration statements, and thus the number of iterations is determined in advance.

Of course it must be noted that a statement in the set of iteration statements can itself terminate the iteration by (for example) transferring control to a statement outside the so-called for-loop in Algol or the DO-loop in Fortran. In this case the iteration terminates before the controlled variable has taken on all its values. Good programming practice, however, requires that such transfers be avoided if at all possible; usually, they can be in languages with a **while** structure, such as in Algol.

A. RALSTON

COPYRIGHTS AND PATENTS, COMPUTER ASPECTS OF

For articles on related subjects see **LEGAL PROTECTION OF SOFTWARE**; and **PROPRIETARY PROGRAM**.

For article on related term see **INFORMATION SYSTEMS**.

Patents and copyrights can be defined as legal devices, incorporating a bundle of intangible, proprietary rights and privileges granted to the authors of certain literary or other "written" expressions for

limited times or to inventors of devices or manufacturing processes. They invest the applicant with the exclusive right to license others to make, publish, vend, and use copies of those original expressions in the case of copyright (Nimmer, 1974), and the right to exclude others from making, vending, or using an invention in the instance of patent. It should be noted that a patent does not necessarily give the inventor a restrictive right to "practice" his invention himself; it grants only the privilege to exclude or allow others to construct and market on his, the patent holder's, terms (Bowman, 1973). Similarly, although it is possible that the copyright holder may not be able to copy or perform his own work or to prevent others from utilizing his creation without his permission, only he can license others to utilize the work.

Thus, we can say that the limited rights granted the copyright holder and the patentee are **exclusionary** rather than permissive. The questions thus faced for the computer professional tend to be **recursive**: What effect does copyright and patent have upon his input and output data; and what effect does copyright and patent have upon his programs and algorithms (Duggan, 1972)?

To answer these questions, it is helpful if one begins by analyzing the purpose served by copyright and patent. In Anglo-American jurisdictions, it is clear that the purpose is to obtain certain public benefits so as "to encourage people to devote themselves to intellectual and artistic creation." The primary end is not to reward the individual author or inventor, or those who invest in the exploitation of those creative expressions, but rather to grant a limited monopoly as a means of promoting "the progress of science and the useful arts" (U.S. Constitution, Art. I, Sec. 8, I. 8).

Means of Copyright and Patent Protection

COPYRIGHTS. It is useful to note that there are now three separate schemes of rewarding authors, subsumed under the notion of copyright. First and foremost, the federal government by positive federal legislation can provide substance to the bare bones of the constitutional grant. Although the federal government enforces some aspects of copyright directly, through its criminal and customs sanctions, the main right given by federal statute is the right to pursue nonlicensed users of protected material; i.e., federal law allows "bounty hunting" for infringers and admits contests in federal courts.

The remedies accruing to the successful plaintiff

can be quite satisfactory, since he is able to claim costs of litigation, actual damages suffered, and the profits realized by the defendant, as well as punitive or exemplary penalties awarded by the court where appropriate. In addition, the infringing products may be impounded while the infringement action is pending, and once the infringement has been judicially determined, they may be destroyed.

If actual damages sustained or profits on the infringement cannot be proved or shown to exist, the trial court may in its discretion award statutory damages "in lieu of actual damages and profits." These damages, if awarded, must be set between a \$250 minimum and a \$5,000 maximum, with the maximum suspended in certain circumstances.

In addition to damages and injunctive relief, the successful copyright plaintiff may also be entitled to full costs of the suit as well as a reasonable attorney's fee. On the other hand, a successful defendant may also recoup costs and a reasonable attorney's fee.

However, in order to obtain these federal rights, one must comply with certain rigid requirements. These are that the work be published (1) with a mandatory notice, (2) in proper form, and (3) in the proper place. The required notice consists of the word "Copyright" or "Copyr.," or the symbol ©, accompanied by the name of the proprietary owner. If the work is a sound recording or a printed literary, musical, or dramatic work, the year of publication must also be included. If a computer program is considered a printed literary work, it would also require the "year" portion of the notice. If a work is considered a book or other printed publication, the notice must be placed on the title page or the page immediately following the title page. Determination of the "title page" of a computer program, even of one utilizing a virtual memory schemata, might prove vexatious. Failure to comply with each and every requirement, generally speaking, will place the published work in the public domain. And once a copyright is lost by failure to comply with these conditions, it may not be regained.

Federal statutory copyright of a published work is obtained by publication with proper notice of copyright, and the subsequent registration and deposit of copies (2) of the work with the Register of Copyrights, Copyright Office. Although this procedure has great significance, it does not create the copyright but merely records it. However, it should be noted that recording and deposit of copies of the published work does not require legal skills and is easily accomplished by direct application to the Copyright Office with the necessary fee (\$6.00 in 1976).

COPYRIGHTS AND PATENTS, COMPUTER ASPECTS OF

Although of lesser importance than federal statutory copyright, two other protection systems should also be mentioned: that afforded unpublished works (the so-called *common law* copyright) and that provided by state legislation, concurrently with federal protection.

Protection for unpublished works, even though based upon general common-law notions as well as state statutes, gives the owner the right to prevent unauthorized copying, printing, vending, publishing of an initial copy, making other versions of the work, performing, and recording the work; i.e., the rights are at least co-extensive with those under the federal copyright statute. Further, if a work is ineligible for statutory copyright because it is not a "writing," it may still receive protection under this co-extensive theory. However, many observers have theorized that the so-called *common law* copyright is in reality a species of tort law directed against those who unfairly appropriate values created by another.

In a recent case (*Goldstein v. California*, 412 U.S. 546, 1973) the Supreme Court enunciated the principle that states possess concurrent jurisdiction with the federal government as long as state legislation does not conflict with federal legislation; i.e., the states may protect those works for which Congress has failed to provide protection unless Congress has deliberately excluded them. This decision contrasts strongly with earlier decisions in patent law wherein the Supreme Court totally voided the concept of unfair competition ("copying") as conflicting with the exclusive jurisdiction of federal law over *patentable* subject matter.

Patents. In contrast to the copyright species of protection where the federal government essentially performs only the ministerial functions of recording and registering works for which protection is sought, an applicant must be the first to achieve a given result before patent protection can be obtained. The Patent Office, to whom an application is made, undertakes a rigorous search of the particular field and grants a patent only if the inventor's claim to be first is verified. Further, unlike copyright, the subject matter of patent is quite limited, including devices, processes, and compositions of matter, but excluding such discoveries as laws of nature, mathematical formulas, or business plans. If the discovery is to be given patent protection, it must not only be novel and useful, but must also be nonobvious to one skilled in the art.

The patent grant is exclusively a civil "bounty" license, with no federal criminal sanctions, and only limited customs barriers. However, it grants the

patentee exclusivity for 17 years; i.e., other persons or companies may not replicate the patent holder's discovery even though they do so independently without access to his invention.

Since the patent search for prior discoveries is particularly arcane, it is doubtful that an inventor could adequately describe his invention without recourse to extremely specialized (and relatively expensive) legal assistance. Further, there is a substantial delay between the time that an application is submitted to the Patent Office and the final determination by it as to whether a patent should be granted. Then, too, the "hunting license" that the Patent Office grants is not treated with extreme respect by many courts; e.g., in one circuit court, not a single patent has been found valid in almost a decade; overall, less than 50% of all challenged patents are upheld on appeal.

At the present time it is arguable whether the states may enact patent-type legislation to protect discoveries that are not subsumed under federal standards. In 1964, the Supreme Court ruled that state laws which clash with the objectives of federal patent laws would not be allowed to exist (*Sears Roebuck v. Stiffel*, 376 U.S. 225: state laws prohibiting the copying and marketing of unpatented articles were unconstitutional). However, in 1974, the Supreme Court held that state laws which provided an *alternative* means of legal protection to federal patent statutes were constitutional (*Kewanee Oil Co. v. Bicron Corp.*, 94 S.Ct. 1879: state trade-secret laws which preclude illicit copying but not independent discovery or "reverse engineering" were not necessarily preempted by federal patent laws).

Copyright and Patent Protection for Computer Programs. In 1964, the Copyright Office announced somewhat reluctantly that it would accept computer programs for registration; however, it clearly indicated that whether a program is a "writing of an author" was a doubtful question.

Notwithstanding these substantial doubts, almost all purveyors of software have chosen to copyright at least part of their program libraries. IBM Circular 120-2083-1 (Rel. 1969) gives detailed instructions to its personnel regarding the procedures involved in licensable programs that are to be copyrighted (Bigelow, 1975).

On the other hand, the Supreme Court ruled in 1972 that computer programs were not patentable (*Gottshalk v. Benson*, 409 U.S. 63). However, the Court of Customs and Patent Appeals has indicated that it considers this decision not applicable to all computer programs, just those in which the algo-

rithm seems to be exclusively mathematical in nature. Whether this narrow interpretation in the Supreme Court decision is justified is at best arguable.

Although computer programs may not be patentable, it is also possible that the courts might decide that programs are not "writings," thus leaving them without any statutory or common-law protection whatsoever. A discussion of the rationale behind this caveat is beyond the scope of this article; however, the possibility of such a "no-man's land" for computer programs should not be dismissed out of hand (Galbi, 1972).

Although no state has yet given explicit copyright statutory recognition to computer programs, unlike the sound recording ("tape and record piracy") field, one should recognize that such enactments could be legislated if a need should be manifested.

Even if computer programs are properly the subject of copyright protection, the extent of that protection may be somewhat limited; e.g., it probably would not extend to the ideas embedded in the program nor to the techniques used in developing or making the program, but only to the format. Even in that case, it is possible that the protection would not extend to the use of the program within the computer but only to the copying of the program for resale to others (Duggan, 1972).

Similarly, if Congress should grant patentlike protection to computer programs, as IBM has advocated (Bigelow, 1972), it is possible that the protection would not extend to the concepts or new principles, but only to the specific series of executable instructions deposited with the Patent Office.

Copyrighted Material as Data for a Computer-Based Information System. Here the problem faced by the computer user is the converse of that posed earlier, in that he wishes to use someone else's copyrighted material in his own information system without the copyright owner's permission (Duggan, 1972, p. 82). There are no legal precedents on which to base freedom of use, and in view of the uncertainty that the Supreme Court has interjected by its Goldstein decision (Goldstein v. California, 1973), any attempt to reason by analogy is at best most hazardous.

Nimmer (1971, §141) points out that there are two elements that must be established if a plaintiff is to be successful in obtaining relief from the defendant's activities: (1) ownership of the copyright by the plaintiff; and (2) "copying" by the defendant.

Assuming that these two elements can be proved where necessary ("copying" usually will be proved circumstantially by showing access and substantial similarity), then the infringer may be able to demonstrate that his "copying" was justified, most notably under the equitable and nonstatutory doctrine of "fair use." The issue of fair use is probably the most awkward and troublesome in the whole field of copyright. Among the elements felt to be applicable in determining whether a particular "copying" is not actionable are: (1) extent and relative value of the copyrighted material and the effects upon distribution; (2) nature and objects of selections made; (3) degree to which the copying prejudices the sale or profits of the original works, etc. Nimmer (1971, §145) argues for a rather simple "litmus" test: Does the "copying" diminish or prejudice the potential sale of the plaintiff's work? Another author (McDonald, 1962) has given a "golden rule" description of fair use: "Take not from others to such an extent and in such a manner that you would be resentful if they so took from you."

In one recent case on this subject (*Williams & Wilkins v. United States* 95 S.Ct. 1344 (1975); judgement of Court of Claims upheld by equally divided court) the court did not hold in favor of the plaintiff's claim that copying activities at the National Library of Medicine had infringed its copyrights. However, this case gives little clue to the boundaries of fair use in current library practice in copying particular articles for substantial distribution upon reader request. However, one may still conclude that at *present* inclusion of copyrighted material within a computer-based full or partial text information retrieval (IR) system would constitute fair use, due to *present* technological and economic realities; i.e., because computer-based IR systems are substantially more expensive than manual alternatives relative to the cost of reproducing the data. Depending on changes in technology, this may not continue to be the case (Duggan, 1972, pp. 82-83).

Conclusions. Although copyrights and patents have had no measurable effect upon computer developments, the lack of firm guidelines as to what can and cannot be done to protect programs, as well as to incorporate protected material in a data base, has lent a substantial degree of legal uncertainty to this industry. It does not appear reasonable to assume that this uncertainty will soon be dispelled, even if Congress enacts the long-awaited Copyright Revision Bill. Instead, it seems quite likely that this "product of censorship, guild monopoly, trade regulation statutes and misunderstanding" (Patterson,

COROUTINE

1968) will continue to cause legal debate and operative uncertainty.

REFERENCES

1962. McDonald, J. "Non-Infringing Uses," *Bulletin Copyright Society*, Vol. 9, Nos. 466, 467.
1968. Patterson, L. R. *Copyright in Historical Perspective*. Nashville, Tenn. : Vanderbilt University Press, p. 229.
1972. Bigelow, R. *Computer Law Service*, Appendix 4-5a. Chicago: Callaghan and Co.
1972. Duggan, M. A. "Viewpoint of the Computer Scientist," in A. Kent and H. Lancour, *Copyright: Current Viewpoints on History, Laws, Legislation*. New York: R. R. Bowker, pp. 66-103.
1972. Galbi, E. W. "Copyright and Unfair Competition Law as Applied to the Protection of Computer Programming," in R. Bigelow, *Computer Law Service*, §4-3, Art. 1. (For a contrary view.)
1973. Bowman, W. S., Jr. *Patent and Antitrust Law. A Legal and Economic Appraisal*. Chicago: University of Chicago Press, p. 2.
1974. Nimmer, M. B. *Nimmer on Copyright*. 2 vol. New York: Mathew Bender, §§100, 141.
1975. Bigelow, R. *Computer Law Service*, Appendix 4-36. Chicago: Callaghan and Co.

M. A. DUGGAN

COROUTINE

For articles on related subjects see **SUBROUTINE**; and **PROCEDURE**.

For articles on related terms see **BUFFER**; **COMPILER**; and **OPERATING SYSTEMS**.

A coroutine, like a subroutine, is a kind of program module. A program module consists of (1) a fixed part (such as instructions in a computer language) that is the same for all instances of execution of the module, and (2) a variable part (such as data and control information) that may vary with different instances of execution. This variable part is called an "activation record," and clearly there is a different activation record for each instance of execution.

A coroutine differs from a subroutine in that the lifetime of a particular activation record is independent of the time when control enters or leaves the

module. Moreover, the activation record of a coroutine maintains a local instruction counter so that, whenever control enters the module, execution begins at the point where it stopped when control last left that particular instance of execution.

These features facilitate quasi-parallel execution. Given a group of coroutines, the execution of each proceeds essentially independently of the rest, subject only to explicit synchronization requests. Execution is only quasi-parallel, rather than actually being parallel, in that only one processor is envisioned (this also avoids timing problems). When a coroutine is selected for dispatch and execution begun, it continues to execute until control is explicitly relinquished by some natural break, whereupon some other coroutine is selected for dispatch.

This kind of quasi-parallel execution is extremely useful when writing programs that consist of several cooperating processes. This arises in discrete event simulation, and also in other event-driven activities such as in a computer operating system or in a multiphase compiler.

For example, the original application of coroutines was to a multiphase compiler. Each phase was written as a coroutine, and the output buffer of one phase was the input buffer of the next. Once given control, a phase would run until its input buffer was exhausted or its output buffer was full, and then would yield control to another phase. In this way the phases ran in quasi-parallel, subject only to the constraint that input for a phase be available, or that output be consumed by a subsequent phase. This unconventional implementation controls buffer size, permitting the use of in-core buffers and resulting in faster compilation.

M. GENTLEMAN

COURSEWRITER. See **AUTHORING LANGUAGES AND SYSTEMS**.

CREDIT SYSTEM APPLICATIONS

For articles on related subjects see **ADMINISTRATIVE-BUSINESS APPLICATIONS**; and **POINT-OF-SALE TERMINAL**.

In early Greece and Rome, and even up through the Industrial Revolution, a creditor did not need a

computer to furnish him with information about a potential debtor. He was already familiar with the applicant's character, his ability to repay the debt, and the type of collateral available to secure the debt. Today, with hundreds of millions of credit cards in existence and billions of credit transactions occurring annually, most credit systems require the use of a modern, high-speed computer.

Following the introduction of oil company credit cards in the late 1930s, their use soon became widespread, making personal knowledge of potential debtors virtually impossible. Current estimates of the number of credit cards in existence in the United States range from 100 to 300 million cards. Any attempt to handle this number of credit accounts without the use of large, high-speed computers would certainly have met with economic failure. Other types of credit systems for bonds and mortgages are also being computerized, but their absolute volumes are small compared with the number of transactions processed through credit cards. Today, most banks will process small consumer loans only through a bank credit card account. In general, a credit card can be used to purchase almost all retail goods.

Computers and Credit Systems. An effective credit system must be concerned with the entire spectrum of credit activities: credit approval, sale authorization, billing, dunning, and collection.

CREDIT APPROVAL. The credit process usually begins when a person files a written application on which he states his name, address, telephone number, employment history and job statistics, bank references, and names of other creditors. An example of a credit card application is shown in Fig. 1. In evaluating the application, the creditor often uses a technique called "point scoring," i.e., assigning a numeric value, depending upon the applicant's response, to each item of information requested. The score or sum total of all points assigned to a particular application is then compared with predetermined values to decide whether to extend credit. Each credit-issuing company that uses a point-scoring technique will include the categories, variables, and assigned scores it has found to be most significant to its own operation.

After data from an application is entered into a computer, points are assigned by an evaluation program. The score for an application is compared with predetermined values to determine what action to take. Sophisticated computer programs have been developed to analyze past experiences, using particular techniques, and to recommend future changes

in an attempt to reduce the cost of selling on credit. It would be very difficult to process the large number of credit card applications today without such a technique.

In developing the account number to be assigned to an applicant, the use of a check digit is usually employed. The computer creates the check digit by using the first $N - 1$ digits of the account number as input to a check-digit generation routine. The resulting calculations produce a single, unique digit, which becomes the last digit of the account number. An example of the "mod 10" technique for generating check digits is given in Fig. 2. The oil company credit card displayed utilizes an 11-digit account number, with the first ten digits (123 456 789 0) being the actual account number and the eleventh digit (3) being the check digit. In order to generate the check digit, using the "mod 10" technique, we start from the low-order position and work toward the high order (from right to left). First, we multiply the odd-position numbers by 2, carrying and adding where appropriate:

$$\begin{array}{rcl} 0 \times 2 & = & 0 \\ 8 \times 2 & = & 16 \quad (\text{carry } 1) \\ 6 \times 2 & = & 12 + 1 \quad (\text{carry } 1) \\ 4 \times 2 & = & 8 + 1 \\ 2 \times 2 & = & 4 \end{array}$$

We now divide each result by 10 and sum the remainders;

$$0 + 6 + 3 + 9 + 4 = 22.$$

Next we sum the even-position digits (from left to right):

$$1 + 3 + 5 + 7 + 9 = 25,$$

and sum the two results:

$$22 + 25 = 47$$

The number that must be added to make this result evenly divisible by 10 is the check digit; in this instance it is the digit 3 ($47 + 3 = 50$). The use of a check digit helps reduce errors in the transcription of account numbers.

SALE AUTHORIZATION. After credit has been approved and a credit limit has been established, each sale above the preset dollar amount must be authorized. The authorization process determines whether the potential purchaser is actually allowed to make purchases on the account, whether the

FULL NAME OF APPLICANT FIRST, MIDDLE INITIAL, LAST				SOCIAL SECURITY NO.		NO. OF DEPENDENTS		NAME OF SPOUSE		DO NOT WRITE IN THIS AREA	
NUMBER & STREET				TELEPHONE				DIST.		ACCT. NO.	
CITY & STATE				ZIP CODE		AREA CODE NO.		APPROVAL		DATE	
HOW LONG AT THIS ADDRESS? YEARS MONTHS				CR. CODE CR. CLASS COLL. CODE				NUMBER OF TRAVEL CARDS DESIRED			
HOME ADDRESS				OPERATOR'S LICENSE NO.				HOW LONG AT PREV. ADDRESS		YEARS MONTHS	
AGE		SINGLE		DIVORCED		BUYING HOME		RENT HOUSE		BOARD	
MARRIED		WIDOWED		OWN HOME		RENT APT.		OTHER		NUMBER OWNED	
EMPLOYER (OR ARMED SERVICE DUTY STATION)				TYPE OF BUSINESS				HOW LONG AT PRESENT JOB? YEARS MONTHS			
BUSINESS ADDRESS				POSITION (OR MILITARY RANK & SERIAL NO.)				APPROXIMATE MONTHLY INCOME			
IF LESS THAN ONE YEAR AT PRESENT JOB, GIVE PREVIOUS EMPLOYER AND ADDRESS								\$			
IS SPOUSE EMPLOYED? <input type="checkbox"/> YES <input type="checkbox"/> NO				SPOUSE'S EMPLOYER				SPOUSE'S POSITION			
• (SHOW • RANCH OR ADDRESS)				SHOW TYPE OF ACCOUNT				REGULAR <input type="checkbox"/> CHECKING SPECIAL <input type="checkbox"/> CHECKING SAVINGS <input type="checkbox"/> LOAN			
FINANCE COMPANY		NAME & ADDRESS						ACCOUNT ACTIVE <input type="checkbox"/>		ACCOUNT CLOSED <input type="checkbox"/>	
CREDIT REFERENCES LIST OTHER CREDIT ACCOUNTS											
OTHER GASOLINE CREDIT CARDS AND ACCOUNT NUMBERS		NAME & ADDRESS						ACCOUNT NUMBER			
		NAME & ADDRESS						ACCOUNT NUMBER			
OTHER CREDIT ACCOUNTS		NAME & ADDRESS						a 30 DAY c 1 INSTALLMENT			
		NAME & ADDRESS						30 DAY c 1 INSTALLMENT			
		NAME & ADDRESS						c 1 30 DAY <input type="checkbox"/> INSTALLMENT			
I AGREE THAT: (1) THIS APPLICATION IS SUBJECT TO CREDIT INVESTIGATION AND APPROVAL. (2) ALL CHARGES WILL BE PAID UPON RECEIPT OF MONTHLY STATEMENTS INCLUDING FINANCE CHARGES ON PAST DUE AMOUNTS. AND (3) CREDIT CARDS ARE SUBJECT TO THE TERMS AND CONDITIONS DISCLOSED WHEN ISSUED.										DATE OF APPLICATION	
SIGNATURE OF APPLICANT										80X-744-2 A	

Fig. 1. Typical credit card application.

account has exceeded its credit limit, and the payment history for previous purchases.



Fig. 2. Mod 10 method of generating credit card identification digits.

Computers are playing an increasing role in the area of sales authorization, and several service bureaus now provide sales authorization services. When contacting a bureau, a merchant subscribing to such a service might communicate directly with the computer by means of his terminal or by phone with a person who has access to the computer. The merchant provides the customer's account number, the amount of the potential sale, and his own location and identification number for service-billing purposes. The sales information is then evaluated according to information in the customer's file, and the sale is approved or rejected. Of course the merchant has the final say and can override the computer's suggestion, but experience shows this is rarely done.

If a high-speed computer is used, the entire authorization process can be accomplished in less than a minute—often within a few seconds—depending on the technique used for communicating with the computer.

Several of the larger credit card-issuing companies are now applying a dark strip of magnetic recording material on the back of their credit cards, similar to those used by San Francisco's Bay Area Rapid Transit system (BART) on their commuter tickets. When the magnetic data strip is used in conjunction with point-of-sale terminals (i.e., a cash register type of terminal connected to the authorization computer), the credit-available field of the data strip can be decremented each time the customer makes a purchase. Many banks now use this technique to make cash available to depositors after banking hours. Both the American Bankers Association (ABA) and the International Air Transport

Association (IATA) have developed standards for recording data on the strip.

BILLING. The billing data may be captured automatically at the time of the transaction, such as when making a credit-card telephone call or when purchasing merchandise at large department stores. These systems generally use the descriptive billing method, in which only an itemized list of charges is sent to the customer for billing purposes.

In systems where copies of the original sales slips are returned to the customer with his bill (country club billing), such as gasoline or food purchases on credit cards, the sales data is usually entered subsequent to the sale. Country club billing systems often use a special-purpose computer with optical character readers to read the sales data from the sales ticket and sort the tickets for return to the customer.

DUNNING AND COLLECTION. This final phase of the credit process occurs only when a debtor is delinquent in his payment. A computer-processed analysis of all active accounts will single out those accounts for which required payment was not received within the specified time period. Depending on the size of the balance due and the amount of time the account is past due, an appropriate dunning letter can be selected from those stored in the computer's memory, printed, and then sent to the customer. After the proper series of dunning letters has been sent, the account can be flagged for personal review or for collection procedures.

As a result of the Truth in Lending Act (passed in 1972), billing, dunning, and collection procedures have become increasingly more complicated. Where a large number of accounts are involved, it would be impossible to determine and report the interest paid by each customer without the capabilities of a modern computer.

A Typical Computerized Credit Card System. A credit card system is a large data base system, typified by a large number of accounts, random access storage, and strictly specified formats for inputting data and generating output reports. Although real time inquiry into the file is usually available, inputting of data, file updating, and printing of invoices and statistical reports are usually done in a batch mode on a scheduled basis.

The basic functions performed by different credit card systems are very similar; however, there has been little sharing or purchasing of existing credit management packages. Each company has tended to develop and write its own in-house credit system, most often using Cobol and a machine

CREDIT SYSTEM APPLICATIONS

language to handle the inquiry portion of the application.

As an example, we consider a typical credit card system used by a petroleum company to process its credit card sales, which handles about five million accounts. At the service station level, it requires a dollar-amount imprinter capable of transferring the customer data from the credit card and printing the dollar amount of the sale on the basic three-part paper sales document. The sales tickets are processed in one location, where the data is captured and entered automatically into the computer system through magnetic tape. The tape is created by a computer with optical capabilities for reading the imprinted information from the paper sales ticket. At the initial reading, the customer's account number and the dollar amount of the sale is sprayed in ink, using a unique recording code consisting of vertical lines and spaces, onto the back of the ticket to assist later in sorting.

Since all accounts are not billed on the same day of the month, the sales tickets are stored in a vault until a customer's cycle is billed. **Cycle billing** is used to level the daily processing time required by the system and to provide a more favorable cash flow situation.

After being billed, a customer is requested to return a portion of the bill with his payment. The payment is sent to a post office lock box, where collections are gathered and processed by the credit card company itself or by a bank, which charges a service fee.

In processing the payments, the amount of the check must be compared with the amount on the returned portion of the bill. Any discrepancy, either shortage or overpayment, requires additional handling and processing time. The checks must be encoded with the magnetic ink characters representing the amount for which the customer's check is drawn. The sum of the checks deposited is compared with the sum of the returned portion of the bills, to keep the system in balance.

For a system of about five million gasoline credit card accounts, between one \$1 and \$2 million per day must be processed and applied to the appropriate customers' accounts in order to keep the master file current. A staff of 75 customer service representatives is needed to process customer inquiries concerning bills, payments, and special charges, and to handle the collection of delinquent accounts. The work of the service representatives is supported by 25 CRT terminals capable of inquiring into the computer's customer file, 10 microfilm readers to examine past history files, and 20 clerical

and typing assistants.

Names and address changes to the file, special handling of certain accounts, and entering data from sales tickets that could not be read automatically by the retina requires another 40 data entry operators. These operators use key-to-disk, key-to-tape, and keypunch machines to enter the data into the computer's mass storage.

The main processing computer is a third-generation computer and has 5 12K bytes of memory, 1 ¼ billion bytes of random access storage, 9 tape drives, 2 card readers, 1 card punch, 35 CRT terminals, and 3 high-speed printers. It rents for \$60,000 per month and requires two operators per shift. In total, the credit system costs \$20 million per year, of which \$4 to \$8 million is charged to bad debts and fraud, \$1.5 million represents postage costs, and the remainder is overhead and payroll cost of the 400 people required to perform its various functions.

REFERENCES

- 1972. Hendrickson, Robert A. *The Cashless Society*. New York: Dodd, Mead & Company.
- 1971. Whiteside, Conon D. *EDP Systems for Credit Management*. New York: Wiley-Interscience.

J. K. AMSBAUGH

CRIME AND COMPUTER SECURITY

For articles on related subjects see **COMPUTERS AND SOCIETY; DATA SECURITY; and SECURITY OF COMPUTER INSTALLATIONS, PHYSICAL.**

For articles on related term see **DATA COMMUNICATION NETWORKS.**

The need for security measures to protect computers and their contents was recognized from the beginnings of computer usage for classified government projects and data. This was easily accomplished for early batch-operated computers that were made to perform one job completely before going to the next.

Problems arose when technical advances allowed the sharing of resources by more than one job at a time, on-line storing of data from job to job, and on-line servicing of more than one user at a time. At the same time, computers were being used for more purposes where the possibilities for unauthorized or

antisocial acts could result in gain, and errors could result in serious losses. Losses from errors were real, and resulted in the need to control access to computers, access to stored data, and access by one program into the memory space assigned to another.

Techniques were developed to protect the computer facilities, use secret passwords and authorization tables for access to computers and data files, and provide programs and hardware features to partition memory. This seemed to control the error problem, but limitations in completeness of computer security became evident when the possibilities of intentional unauthorized acts were considered.

The first computer-related act involving criminal prosecution occurred in 1964. A programmer in Texas stole his employer's computer programs, worth \$5 million, and was convicted and served a five-year prison sentence (Texas v. Hancock). The first federal crime involving the use of a computer occurred in 1966 when a programmer in a bank put a change in a program to ignore his checking account when checking for overdrafts (U.S. v. Bennett). In 1968 an accountant was caught after embezzling \$1 million over a six-year period, and was convicted and sentenced to ten years in prison. He used a computer to simulate the operation of his company to assist in regulating his peculations to avoid detection (California v. Mansfield). The first crime of stealing a program, held as a trade secret, from the memory of a computer through a remote terminal and telephone circuit occurred in 1971 (California v. Ward).

Broadening the concept of computer crime to any kind of act associated with computers where the victims suffered or could have suffered losses, or where the perpetrator experienced or could have experienced gain, provides a more all-inclusive perspective on the computer security problem. The number of such recorded acts has increased from 8 in 1969 to 23, 47, and 39 occurring respectively in each subsequent year.

A collection and study of 160 reported cases (Parker et al. 1973) produced valuable facts for developing more effective computer security. In the cases studied, it was found that computers play four roles:

1. *Subject*: Acts such as vandalism or unauthorized usage attack computers or their contents.
2. *Unique environment*: This refers to acts involving computer programs and data in computer-readable form.
3. *Tool*: The computer is used to plan or carry

out a crime.

4. *Symbol*: Fraudulent acts, deceit, or intimidation are charged as due directly to the use of computers, such as in false advertising of computer dating services or abusive dunning of consumers for unpaid bills.

The population of potential perpetrators tends to be in computer-related occupations, for these are the only people with the necessary skills, access, and knowledge to perform the acts. Perpetrators in other occupations need the help of computer people to accomplish their ends. In fact, one study comparing computer- and noncomputer-related bank embezzlements bears this out. Collusion was present in one-third of the computer-related cases and in less than 4% of the other cases.

Perpetrators are generally young, energetic, highly motivated, intelligent people. In some cases, their acts deviate in only small ways from accepted practices of the perpetrators' associates. The Robin Hood syndrome is common: rationalizing acts against organizations, but believing acts against individuals to be immoral. This is combined with the vending machine syndrome-the challenge in the game of beating the machine and the absence of established ethical standards in the new computer-related occupations. A personal philosophy conducive to unauthorized acts and to criminal acts is created.

Assets subject to loss in computer-related acts are different enough from those in previous manual systems to confound the legal profession, the auditors, and traditional security people. Computer programs and data files represent entirely new types of assets-assets that are being subjected to theft, fraud, unauthorized use, and use in extortion. Data stored magnetically and electronically are compact, volatile, only indirectly accessible, and highly time-dependent in value compared to paper-based data in previous manual systems. This produces a different environment for the criminal and offers real possibilities for effective protective measures because of the more structured and controlled environment in computers. In fact, a contest is in progress between the criminal and security organizations when computers are used to penetrate the crime-prone environments. It also appears that the stakes are high; losses noted among the reported computer-related crime cases are ten times higher than in comparable noncomputer related crime.

A practical objective of security (Parker et al., 1973) is to reduce the number of potential perpetrators to as few people as possible. Technical

CRITICAL PATH METHOD

methods to accomplish this include partitioning computer memory, imposing authorization requirements on use of programs and access of programs to stored data, using cryptographic techniques, and identifying people and outside processes attempting to gain entry to the system. Important principles imposed on the security design of such systems include the following:

1. *Absence of design secrecy*: Assume the potential attackers will know as much about the system and its security features as do its designers.
2. *Least privilege*: Provide no more privilege to a process within the system or to users of the system than is necessary to allow them to adequately accomplish their authorized purposes.
3. *Compartmentalization*: Minimize the penetration and subversion of other processes when one process is penetrated or subverted.
4. *Threat monitoring*: Provide detection mechanisms for all anticipated penetration and subversion methods, and produce a reporting facility for monitoring and auditing.
5. *Secure kernel*: Isolate the security processes from the system with minimal and formal interfaces to the system.
6. *Auditability*: Structure security processes to allow complete auditing for integrity.
7. *Permissive authorization*: Base authorization on permission to use any system function rather than exclusion from selected functions for all system activity.
8. *Acceptability*: Make security restrictions acceptable to personnel.

A computer system will be no more secure than its environment or the procedures used in its operation and maintenance. Adequate controls over physical access must be established, and sufficient administrative procedures and discipline in maintaining them must be imposed. Commonly accepted practices include manually or automatically guarded doors; coded badges to identify a wearer's authorized access area; separate performance and dual control of sensitive functions; preemployment, in-depth screening of staff; posting appropriate warning signs; and control and labeling of valuable materials and media. Operational security audits should be made periodically, and backup copies of sensitive programs and data should be stored in safe places. Assignments of security responsibility should be made and a contingency plan drawn up and made readily available. Error recovery and computer restart procedures should be formalized.

These security precautions should be present in any computer facility where possible threats that could result in losses, injuries, or damage are present. In the future, many security activities traditionally performed by people should be automated. However, technological methods are necessary, but not sufficient. Current computer systems cannot be sufficiently patched up—except in special cases and at high cost. No comprehensive technological solution is at hand (and none is expected for some years) so that entirely new computer systems will have security included as a design criterion from the beginning. Even if such a solution is found, computers will still be vulnerable to a few systems maintenance people with sufficient skills, knowledge, and access. However, this limited accessibility would be a vast improvement over the present situation, in which large numbers of computer users have the capacity to subvert computers at both the system and application levels, as substantiated by actual experience.

The use of computers in data communication networks poses a new problem for security. Input and output functions are dispersed to remote terminals in relatively informal and isolated environments, making it especially difficult to identify the authorized terminals and terminal users, and to protect data transmission paths. However, these problems should be solvable in the same time frame as other computer security problems.

REFERENCES

- 1967. Ware, W. H. "Security and Privacy in Computer Systems," *AFIPS Proc.*, Spring Joint Computer Conference.
- 1970. Smigel, E., and H. Ross, *Crimes Against Bureaucracy*, Van Nostrand Reinhold.
- 1973. Anderson, J. P. "Computer Technology Planning Study," U.S. Air Force, ESD-TR-73-5 1, Vol. 1.
- 1973. Parker, D. B., S. Nycum, and S. Oura. "Computer Abuse," Stanford Research Institute, Menlo Park, Calif. NSF Grant No. GI-37226 (October).

D. B. PARKER

CRITICAL PATH METHOD. See PERT/CPM.

CRYPTOGRAPHY, COMPUTERS IN

For article on related subject see CODES.

Cryptography is the science that deals with the concealment of information in messages for the purpose of making them secret. It is distinguished from steganography, which deals with the concealment of the very existence of secret messages. The alteration, or encryption, of a message is a **transformation** from the original form of the message into a form that is unintelligible to anyone who does not possess some converting information, called the "key." It is generally assumed that both the original message and its encrypted form, or cryptogram, are composed of sequences of discrete symbols.

There are two distinct methods by which the transformation of a message into a cryptogram may be accomplished: *codes* and *ciphers*. Codes are distinguished by the fact that the basic units of the message to be encoded are selected on the basis of meaningful sequences of symbols, the number of symbols chosen per unit being variable. For example, the basic units for encoding may be the individual words or perhaps even phrases of the message. Ciphers are characterized by the fact that the basic unit of encipherment is always the same. For example, one may encipher single symbols, or possibly groups of symbols of a fixed length.

There are two basic ways in which a message may be enciphered into a cryptogram: by *substitution* or *transposition*. As an example of simple substitution encipherment, consider the following key, or *cipher*, on the symbols of the English alphabet:

Message Symbol:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cryptogram Symbol:	ORGANIZEDBCFHJKLMPQSTUVWXY

The message **BUY GOLD TOMORROW** would have the cryptogram **RTX ZKFA SKHKPPKV**. Each symbol of the message is substituted by its corresponding cryptogram symbol according to the above key.

A well-known transposition encipherment is the rail-fence transposition, in which the message symbols are arranged in one order and the cryptogram is composed by selecting the same symbols in different order; thus

B Y O D O O R W
U G L T M R O

would have the cryptogram **BYODOORWUGLTMRO**.

The process of enciphering a message may be a very complex combination of these basic transformations, thereby making *cryptanalysis*, the task of determining the original message without the aid of the key extremely difficult indeed. A *cryptanalyst*, the person who attempts to break or solve the cryptogram through cryptanalysis, will take measurements of the cryptogram, using such techniques as single-symbol frequency distribution (the number of occurrences of each distinct symbol) or contact variety (the number of distinct symbols a given symbol precedes or follows).

Many times, the great speed of modern computers is used to help the cryptanalyst in this arduous task, since the value of the information contained in the cryptogram is usually quite substantial. Computers have also been used to actually construct cryptograms but more likely than not this task will be accomplished by specialized devices.

In recent years, cryptography has been employed to protect sensitive information contained in computer systems (e.g., on disk or magnetic tape) from being used by unauthorized personnel. Due to the secrecy surrounding the development of both cryptographic techniques and cryptanalysis there has not been much written about the subject. For a historical account of cryptography, the reader is referred to Kahn (1967). A mathematical treatment can be found in Shannon (1949). The development of cryptographic techniques for computers is given in Krishnamurthy (1970).

REFERENCES

- 1949. Shannon, C. E. "Communication Theory of Secrecy Systems," *Bell System Technical Journal*, Vol. 28, No. 4 (October), pp. 656-715.
- 1967. Kahn, D. *The Codebreakers*. New York: Macmillan.
- 1970. Krishnamurthy, C. V. "Computer Cryptographic Techniques for Processing and Storage of Confidential Information," *International Journal of Control*, Vol. 12, No. 5, pp. 753-761.

F. STAHL

CUBE

For article on related subject see **COMPUTER USER GROUPS**.

CUBE is the official organization of the users of Burroughs computers. The name "CUBE" is an

CURRENT AWARENESS SYSTEMS

acronym derived from Cooperating Users of Burroughs Equipment.

The present association was formed in 1962 through the merger of two earlier organizations of Burroughs computer users, CUE and DUO. These two predecessor groups had comprised the users of Burroughs B220 and B205 (Datatron) systems, respectively. The advent in the early 1960s of such computers as the **B200** and **B5000** prompted the merger and assured that a single association would serve the needs of all Burroughs computer users.

In 1971, CUBE was incorporated as a nonprofit association with the full name of CUBE, Incorporated. Currently, membership in CUBE is open to users of the B 1700 or larger computers. These include the **B300/B500**, the **B2700** through **B4700** series, and the **B5500**, **B6700**, and **B7700** systems. At the beginning of 1974, CUBE members represented approximately **700** installations. CUBE membership is international in scope.

CUBE meetings, which last for 3 ½ days, are regularly held twice each year. One of these conferences is typically held in the eastern part of the United States, and the other is usually scheduled for a western city.

The basic structure of CUBE centers around hardware-oriented subgroups (B 1700, Medium Systems, **B6700**, etc.), each having its own elected officers. The semi-annual meetings consist of a series of technical sessions, user panels, workshops, etc., for each subgroup. The sessions are planned by the subgroup officers, and the topics are selected to be of current interest to the members and to be in accord with CUBE's purpose of providing a communications forum for the users of Burroughs computers.

The subgroup sessions are augmented by a series of industry-oriented Common Interest Group meetings whose programs are of special interest to Burroughs users from the financial, manufacturing, hospital, education, government, etc., sectors.

The communication emphasis within CUBE includes not only the sharing of ideas among users, but also the communicating to Burroughs of user sentiment on industry trends, future needs, and current products. Formal, two-way communication procedures have been established to facilitate this aspect of CUBE meetings.

Users of future Burroughs computers are assured of an opportunity to organize themselves into a subgroup within CUBE by procedural mechanisms already present in the bylaws of CUBE.

More information about CUBE may be obtained by writing to the CUBE secretary, Burroughs Corporation, P.O. Box 418, Detroit, Michigan 48232.

The presidents of CUBE have been:

Vic Whittier, Dow Chemical Co., 1962-1963
Rusty Langenfeld, Northern Natural Gas Co., 1963-1964
John Lynn, NASA, 1964-1965
Bill Macomber, Harvard Trust Co., 1965-1966
Pete Jensen, Georgia Institute of Technology, 1966-1967
David Guest, Young & Rubicam, International, 1967-1968
Henry Bowlden, Westinghouse Electric Corp., 1968-1969
John Dorosk, Financial Computer Services, 1969-1970
Bill Eichelberger, University of Denver, 1970-1971
Bob Steffens, Michigan National Bank, 1971-1972
Henri Berce, Marathon Oil Co., 1972-1973
Henry Carter, United Data Centers, 1973-1974

T. S. GRIER

CURRENT AWARENESS SYSTEMS

For articles on related subjects see **INFORMATION RETRIEVAL**; and **MEDLARS-MEDLINE**.

A current awareness system is a system for notifying users on a periodic basis of the acquisition of information (usually literature) by a central file or library, which should be of specific interest to the user. Such systems are designed to respond to the problems of search selectivity and timeliness by carrying out information searches, using only small files of selected documents. User queries, or interest profiles, are typically stored on a permanent basis, to be processed periodically against small files of documents that might be newly received at a given information center. Users are notified on a weekly or monthly basis of new acquisitions that match their interest profiles. Under ideal conditions, such systems for the selective dissemination of information (SRI) are able to retrieve information exactly tailored to meet the specific, possibly changing, needs of each user, while supplying the output directly on a periodic and dependable basis.

The rapid development of selective dissemination services is due to two main factors: First, **SDI**

services are much less expensive to implement than on-demand searches because there is no need to include in the document collections the backlog information covering many years in the past. Second, the existence of the many distribution services of magnetic tape data bases—normally containing titles, citations, and sometimes index terms and abstracts of published articles and research—insures the availability of the needed input data on a regular basis. Normally, the producers of the tape data bases sell the SDI services directly on their own account, or they make the document tapes available to third parties, who in turn provide the dissemination services. Ideally, a given organization might process data bases in many different subject areas and in many languages, thereby rendering flexible services to a large and heterogeneous user population. SDI services are implemented in all areas of applied science and engineering, and in many of the natural and social sciences as well.

It has been conjectured that a flexible SDI service may be the answer to the inefficiencies now inherent in the normal publication system, in which each published item carries high publication costs and minimal readership. An improved, more economical system might then eliminate bound-volume journals entirely and restrict certain types of books to library use, while providing at the same time an

efficient distribution of individual articles and citations that are tailored to specific user populations. In view of the difficulties that still arise in the design of effective dissemination services, and considering the problems that would result for the publishing industry from such a drastic change of publishing standards, such developments may appear to be premature. They must, however, be considered seriously for the not-too-distant future.

A flowchart outlining a typical SDI service is shown in Fig. 1. Specific SDI features are as follows:

1. **Universal features:** utilization of user feedback; automatic or manual profile revision; option for hard copy of abstract and/or full text; and system evaluation.

2. **Optional features:** use of free text (title or abstract) search; use of multilingual thesaurus; searching of multiple data bases; incorporation of preprinted in addition to published information; incorporation of citation, author, or institution alert; and special distribution to designated recipients.

It should be noted that, to improve services at a later time, nearly all SDI services include feedback provisions that utilize user opinions about the effectiveness of the search output. Specifically, response cards are often included with the output sent to the

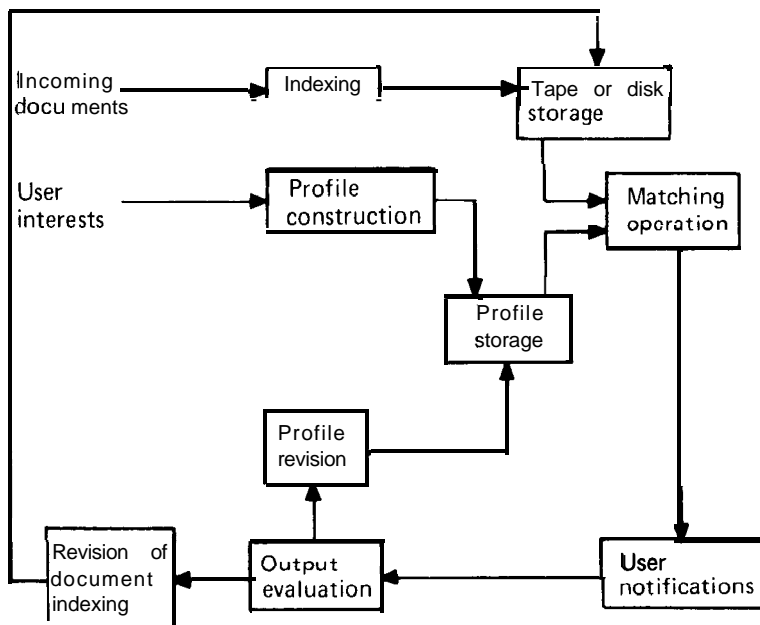


Fig. 1. Typical simplified selective dissemination service.

CURSOR

user population to enable the recipients to return information concerning the retrieved materials. Direct assessments of usefulness are sometimes wanted for each retrieved citation; alternatively, the return cards representing user requests for hard copies of certain retrieved documents are automatically taken by the system operators as an expression of approbation on the users' part.

In either case, the user profile statement may be updated, often manually, by reinforcing or increasing the weight of profile terms that match terms included in retrieved items designated as relevant by the users. Profile terms included in documents identified as nonrelevant may be similarly demoted or decreased in weight. Occasionally, the document indexing may be also changed as a function of user judgment. The corresponding feedback paths are indicated in Fig. 1.

The feedback feature is particularly useful in a research environment where user interests may change fairly rapidly. The profiles can then be adjusted little by little as the users express satisfaction or dissatisfaction with the materials obtained from the retrieval service,

Among the useful optional **SDI** features is the possibility of including in the distribution service those documents in preprint form or other items that are not intended for eventual formal publication. This option provides the means for bypassing the normal publication process and for avoiding publication delays. Delays can also be avoided by having the authors of certain items provide a special distribution list of recipients to whom the corresponding documents are to be sent regardless of the profile-matching results. Finally, the participants in an **SDI** system can gain better service by extending their profiles to include not only subject terms, but also names of authors or of institutions whose documents they wish to receive automatically.

Another extension of optional service permits the inclusion of document citations in the user profiles so that new items citing the original profile documents will be automatically retrieved with other pertinent materials. In its simplest form, such a citation-monitoring system would alert a given participant whenever one of his own papers was being cited by some outside author, assuming that the users of the service include their own documents as part of their profiles. Alternatively, in many circumstances a citation alert system can simplify normal subject searches by eliminating the problems of vocabulary know-how and control that affect document indexing and query formulations.

An evaluation of **SDI** services shows that a large

proportion of the materials retrieved for the user population is indeed germane to user interests. However, complaints arise because of the large volume of output continually delivered by the services. Even if the proportion of relevant items is fairly high, users receiving 30 or 40 citations every week may eventually tire of the system and revert to on-demand searches that furnish output only when specific requests for service are made,

G. SALTON

CURSOR

For articles on related subjects see **COMPUTER GRAPHICS; JOYSTICK; and RAND TABLET.**

A cursor is a special character or symbol on a soft copy (display) terminal, which is used by an interactive program as a pointer or attention-focusing device to allow communication and interaction between the console operator and the program. On alphanumeric displays, for example, a (blinking) underscore or overscore character (or an inverted v, called a "caret") may be used by the program or the hardware to indicate where the next character to be typed by the operator will appear or where the program should start or stop reading the message prepared by the operator at the console. As the operator types, the cursor is automatically advanced, including automatic movement to successive lines. Additionally, many alphanumeric displays have several special keys to move the position of the cursor: left, right, up, down, home (upper left-hand corner of the display).

On a vector-graphics console, the cursor symbol is typically used in conjunction with a manual input device such as a joystick or data tablet to provide the console operator with visual feedback as he moves the joystick or the tablet stylus. The beam deflection applied to the cathode-ray tube to display the cursor is derived from the hardware registers that record the angular displacement or x-y position of the input device to which the cursor is coupled. In effect, the operator can "drive" the cursor around the screen until it is placed at the exact x-y location desired. Once positioned, the operator can indicate to the system that this particular cursor position is to be transmitted to the program. The cursor x-y position can then be used by the program either to provide

raw x-y *drawing* input data or to be compared against x-y positions of information already on the screen for identification purposes (thereby providing a substitute for identification by *lightpen* picking).

A. VAN DAM

CYBERNETICS

For articles on related subjects see **AUTOMATION**; and **CONTROL APPLICATIONS**.

For article on related term see **WIENER**, **NORBERT A.**

Cybernetics is a science founded in the 1940s by a group of scientists and engineers led by N. Wiener and A. Rosenbluth, who coined the word "cybernetics" (from Greek: pilot, steersman, governor) to designate the science of "control and communication in the animal and the machine" (Wiener, 1948). This definition still expresses the substantial content of cybernetics, although there is a broad spectrum of current interpretations (Drozin et al., 1973).

Cybernetic concepts cluster around three related component concepts: systems (animal or machine), communication between systems, and regulation or self-regulation of systems. Since the first two are common to nearly all fields of knowledge, it is the third component, regulation, that distinguishes the discipline. Cybernetics is the science of regulation and control-purposeful regulation for adaptive system survival (Beer, 1966).

Cybernetics borrows ubiquitously from other sciences. Borrowing from mathematical concepts, cybernetics concerns all conceivable *sets* of systems (Ashby, 1970); and from physical and psychological concepts, it "deals with all forms of behaviour in so far as they are regular, or determinate, or reproducible" (Ashby, 1970, p. 1). However, to be of practical interest, cybernetic systems have two properties: (1) some aspect must provide observable data over a period of time (the "protocol," Ashby, 1970, p. 88); (2) from the protocol it must be possible to infer some stable configuration or regularity in transformation of states. Without observable regularity in transformation, a system is said to be unconstrained. Without constraint, it is unpredictable; if it becomes unstable, it cannot be restored to stability or is uncontrollable. For regulation, a system must show some regularity.

Time is the principal cybernetic variable, while

"variety" is the principal dependent variable. Variety is quantitatively measured by the logarithm (usually base 2) of the number of discriminations that an observer (or a sensing system) can make relative to a system (Ashby, 1968, p. 124). For example, in the phrase "take care" the variety is $\log_2 6 = 2.585$ bits if the system is the set of distinguishably different letters; $\log_2 2 = 1$ bit if the system is the set of words; and $\log_2 1 = 0$ if the system is the message considered as a unit. Because variety is based on discrimination of differences, it measures equally well all psychophysical or higher cognitive discriminations (Heilprin, 1973). For example the variety in five psychophysically discriminated shades of green is $\log_2 5 = 2.322$ bits, the same as the variety in a decision process from a choice of five abstract alternatives.

The real significance of variety lies not in its absolute amount, but in the possibility of its increase or decrease. We increase sensory variety when we gather data, decrease it when we summarize, compress, abstract. Both processes are necessary for cognition. However, "lower" cognitive processes are associated with data gathering or increase in concrete sensory variety, whereas "higher" cognitive processes are associated with data condensation, abstraction, or decrease in concrete variety.

When the variety shown by a system under one set of conditions is less than that shown by the system under another set of conditions, the relation between the two sets of variety is a *constraint* (Ashby, 1968, p. 127). For example, suppose two couples (A and B, or four voters) can each vote independently R or D. Then the number of distinguishably different outcomes is $2^4 = 16$ and the variety is $\log_2 16 = 4$ bits. If, however, Mr. A always defers to Mrs. A's judgment and votes the same as Mrs. A, the number of different outcomes is 8 and the variety shown $\log_2 8 = 3$ bits. If, further, Mr. and Mrs. B always vote R, then the variety in the outcomes is $\log_2 2 = 1$ bit. The progressive decrease in variety from 4 bits to 3 to 1 corresponds to increase in constraint on the system showing the variety. Returning to the requirement for regulation—that there must exist some constraint in order to predict the behavior of a system—it is apparent that to regulate a system is to impose a constraint on its variety.

The cybernetics of constrained and unconstrained sets was advanced by the insight of Wiener (1948) who said that "the transmission of information is impossible save as a transmission of alternatives," and by Shannon's observation that "the significant aspect is that the actual message is

CYBERNETICS

selected from a set of possible messages" (Shannon and Weaver, 1949) Thus, regulation implies capability to prevent occurrence of unfavorable alternatives. Therefore, it implies transmission of these alternatives to the regulator, which in turn must respond (with a command message directed toward preserving the stability of the regulated system).

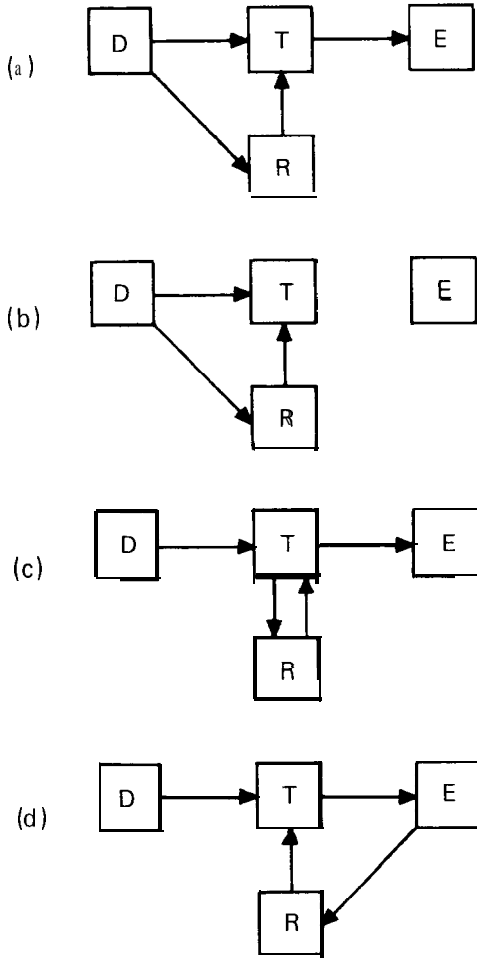


Fig. 1. Law of requisite variety. (a) One-step communication channel regulation: $D \rightarrow R$. (b) Perfect (one step) regulation: $D \rightarrow R$. (c) Two-step regulation: $D \rightarrow T \rightarrow R$. (d) Three-step (error controlled) regulation: $D \rightarrow T \rightarrow E \rightarrow R$.

Fig. 1 shows the basic elements of a regulatory system. The system whose essential variables (E) are

to be kept within certain limits depends on communication of variety (information) between system disturbance (D), the regulator (R), and the environment (T). The most direct regulation is DR, shown in Fig. 1(a). The signal arrives from D in time for R to act on T before T affects E. Figs. 1(c) and 1(d) show paths DTR and DTER, progressively less effective. Perfect regulation [Fig. 1(b)] would leave E isolated from external disturbance-i.e., unaware because of noncommunication that a disturbance had occurred.

A system is said to be well regulated when, through the intervention of the regulator and the environment, a disturbance cannot permanently drive the system from a state in which it is stable (retains its structure and function—"survives"). Lack of regulation occurs when the system is transformed to a state from which it cannot return to a stable state; i.e., cannot survive.

The principal law of cybernetics (credited to Ashby, 1968, p. 206) is the law of requisite variety. This states that if $\log V_d$ is the variety in the possible ways in which a disturbance D can affect a system E (to be regulated by a regulator R), and $\log V_r$ is the variety in R's alternatives (optional ways of response to D), then the variety in the possible outcomes ($\log V_o$) affecting E cannot be forced by R below ($\log V_d - \log V_r$), or $\log (V_r / V_d) > \log V_o$.

This law applies to all forms of regulation, and is independent of field of science or technology or of specific mechanism. Loosely interpreted, it means that—assuming the disturbance, environment, and the system itself are fixed and cannot be altered—the only way to increase E's probability of survival is to increase R's variety (R's versatility, or the number of different modes of response which R can make in order to protect E's stability as affected by D). However, satisfying the law by increasing V_r does not guarantee perfect regulation, i.e., perfect shielding of E. Just as the existence of constraint is necessary but not sufficient for regulation, satisfying the law of requisite variety is necessary but not sufficient for successful regulation (Heilprin, 1973, p. 24).

Space prevents discussion of many prominent cybernetic features such as classification of cybernetic systems by intractability to control (determinate, complex, and "very large"), black-box theory, feedback, isomorphism and homomorphism, and epigenetic theory of regulation. See the references cited and a growing list of periodicals, among which are *Journal Of Cybernetics* (American Society of Cybernetics), *Transactions on Systems, Man and Cybernetics* (IEEE), and *Soviet Cybernetic Review*.

REFERENCES

1948. Wiener, N. *Cybernetics or Control and Communication in the Animal and the Machine*. Cambridge, Mass. : M.I.T. Press (New York: Wiley, 1948; 2nd ed., 1965).
1949. Shannon, C. E., and W. Weaver, *The Mathematical Theory of Communication*. Urbana: Univ. of Illinois Press, p. 3.
1968. Ashby, W. R. *An Introduction to Cybernetics*. London: Methuen (University Paperbacks, 1956), chap. 7.
1970. Beer, S. *Decision and Control*. New York: Wiley, chap. 15.
1973. Drozin, V. G., R. Fisher, F. F. Kopstein, G. Pask, and M. Toda. "What is Cybernetics?," FORUM, American Society for Cybernetics. Vol V, No. 4 (December), pp. 3-8.
1973. Heilprin, L. B. *Impact of the Cybernetic Law of Requisite Variety on a Theory of Information Science*. College Park, Md.; Univ. of Maryland Computer Science Center, Report No. TR-236, March, ERIC No. ED 073 777, pp. 9-10.

L. B. HEILPRIN

stolen by an I/O channel (from the CPU) between two cycles of memory use by the CPU. This is possible and convenient, at least periodically, because the CPU is self-driven-except possibly between some substeps of a process it is conducting -and has no fixed time demands on memory. Furthermore, there are occasions, particularly in simpler CPU designs, where the instruction being obeyed (e.g., division) is processor-limited and memory access is temporarily suspended.

The I/O equipment is, on the other hand, quite different. Its use of the memory, though generally less frequent than that by the CPU, is much more time-constrained. For many I/O devices such as disks and tapes, data is produced or required at fixed intervals. The need for data transfer occurs relentlessly at fixed time intervals. In transferring data from a tape to memory, the previous byte or word must have been stored before the next arrives; otherwise data is lost. This problem is somewhat alleviated by the use of single or multiple buffers in the device controller and/or channel, but in any case there are important recurring time demands for memory access. These can be met by the technique of cycle stealing in those CPU designs in which processor activity is suspended for at most a memory cycle while a memory access is made by the I/O system.

K. C. SMITH AND A. SEDRA

CYCLE STEALING

For article on related subject see **MEMORY**: Main.

Cycle stealing is a technique for memory sharing whereby a memory may serve two autonomous masters and in effect provide service to each simultaneously. One of the masters is commonly the central processing unit (CPU), and the other is usually an I/O channel or device controller. Fig. 1 illustrates a memory cycle (numbers 3 and 5) being

CYCLE TIME

For articles on related subjects see **LOCAL STORE**; and **REGISTER**.

The cycle time of a computer is the time required to change the information in a set of

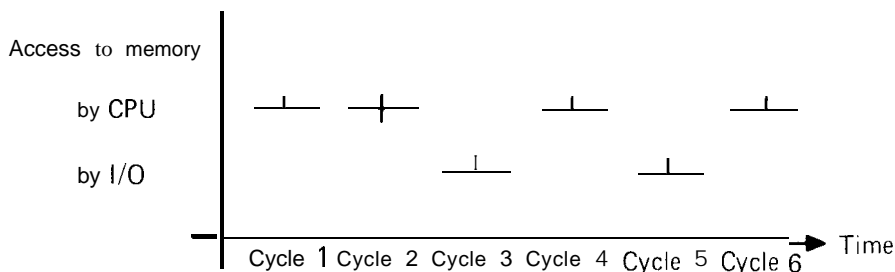


Fig. 1 Cycle stealing.

CYCLIC REDUNDANCY CHECK

registers. This is also sometimes called the “state transition” time.

The register cycle time of a processor is sometimes referred to as the “internal cycle time,” “clock time,” or simply “cycle time”; occasionally, confusion develops between the internal cycle time (referenced to registers) and the main memory cycle time. The memory cycle time is usually several times the internal cycle time.

The internal cycle time may not be of constant value. There are basically three different types of cycle-timing organizations.

1. **Synchronous (*fixed*):** In this scheme all operations are composed of one or more cycles, with the fundamental time quantum being fixed by the design. Such systems are also referred to as “clocked,” since usually a master oscillator (or clock) is used to distribute and define these cycles.

2. **Synchronous (*variable*):** This is a slight variation of the first scheme; certain long operations are allowed to take multiple cycles without causing a register state transition. In such systems there may be several different cycle lengths. For example, a register-to-register transfer of information cycle might take one cycle while a register-to-adder and return-to-register cycle would perhaps be two or three cycles.

The fundamental difference between the fixed and variable synchronous types is that the former stores information into registers at the end of every cycle time, whereas the latter sets information into registers after a number of cycles, depending upon the type of operation being performed.

3. **Asynchronous *operation*:** In a completely asynchronous machine there is no clock or external mechanism that determines a state transition. Rather, the logic of the system is arranged in stages; when the output value of one stage has been stabilized, the logic signals the input at the stage to admit a new pair of operands.

Asynchronous operation is clearly advantageous when the variation in cycle time is significant, since a synchronous scheme must always wait for the worst possible delay in the definition of the time quantum required. On the other hand, when logic delays are predictable, synchronous approaches have an advantage because several additional stages of logic are required in the asynchronous scheme to signal completion of an operation.

In actual practice, most systems are basically synchronous (either fixed or variable) with some

asynchronous operations being used for particular parts of the machine, such as handling access to main memory.

M. J. FLYNN

CYCLIC REDUNDANCY CHECK

For articles on related subjects see **CODES**; **ERROR CORRECTING CODES**; and **PARITY**.

In modern computer systems, data is continuously transferred between the main processor and its peripherals, storage, or terminals. Errors may be introduced during the reading, writing, or actual transmission of this data. Consequently, error control has become an integral part in the design of modern computers and communication systems. The most commonly used methods for error detection involve the addition of one or more bits, called “redundancy” bits, to the information-carrying bits of a character or stream of characters. These redundancy bits do not carry any information; they are merely used to determine the correctness of the bits carrying the information.

Perhaps the most commonly used method for error detection is the simple parity check. Parity may be even or odd, meaning that the sum of the “one” bits of any character, including the parity bit itself, will always be even or odd, depending upon which arrangement is chosen.

Fig. 1 illustrates a form of two-dimensional parity checking used on some magnetic tapes that can detect and even correct some types of errors. The six-bit characters are arranged in columns with a seventh odd parity bit, called the “vertical redundancy check” (VRC), added to make the sum of the “one” bits in each column an odd number. Similarly, an odd parity-check bit, called the “longitudinal redundancy” check (LRC), is added at the end of the block for each row of bits. As the tape is read, the VRC and LRC are regenerated and checked with the check characters read. If equal, the information is assumed correct. If not equal, the block is read again. Some types of errors, like the one shown in Fig. 1, may be corrected by using this method.

Cyclic redundancy checking is a far more powerful error-detecting method. Here, all the characters in a message block are treated as a serial string of bits representing a binary number. This number is then divided modulo 2 by a predetermined binary

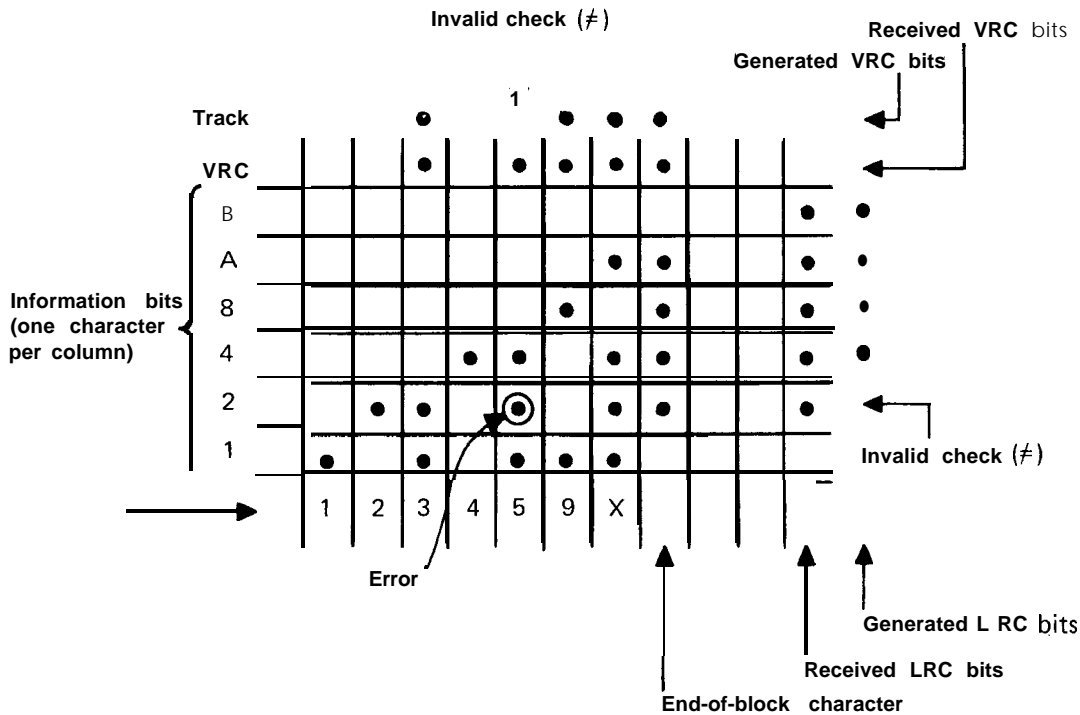


Fig. 1. Error detection using LRC and VRC bits. An extra "1" bit has been introduced in the character "5." Assuming no errors in the received check bits, the error **must** occur at the intersection of the **invalid** check column and row. The error bit must be reversed. In this case, the "1" must be changed to "0."

number and the remainder of this division is appended to the block of characters as a cyclic redundancy check (CRC) character. The CRC is compared with the check character obtained in similar fashion at the receiving end. If they agree, the message is assumed correct. If they disagree, the receiving terminal will demand a retransmission. This is usually called the ARQ (automatic repeat request) method of error control and is very commonly used in data communication. The CRC character is also called the "cyclic check sum," or simply the "check sum" character.

To show how the CRC is generated, let the message consist of k bits, $a_0 a_1 \dots a_{k-1}$, $a_i = 0$ or 1 . Then we form the $(k-1)$ -degree polynomial:

$$M(x) = a_0 + a_1 x + \dots + a_{k-1} x^{k-1} = \sum_{i=0}^{k-1} a_i x^i. \quad (1)$$

If we wish to include r CRC bits, $r < k$, $M(x)$ is multiplied by x^r (this is equivalent to shifting the message bits r places to the right). Let $G(x)$ be another polynomial-called the "generator" or

"checking" polynomial-of degree r , whose coefficients are also 0 or 1 . We divide $x^r M(x)$ by $G(x)$, obtaining

$$\frac{x^r M(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \quad \text{mod } 2 \quad (2)$$

where the "modulo 2" indicates that all sums and differences of coefficients are taken as 0 , if the result is 0 or even, and 1 if it is odd. Thus, from Eq. (2)

$$R(x) = x^r M(x) + Q(x)G(x) \quad \text{mod } 2 \quad (3)$$

where $R(x)$ is the remainder and $Q(x)$ is the quotient. The code word $W(x)$ is

$$W(x) = Q(x)G(x) + x^r M(x) = x^r M(x) + R(x) \quad \text{mod } 2, \quad (4)$$

and what is transmitted are the coefficients of $W(x)$.

Note that $W(x)$, which is of degree $r + k - 1$, contains the original k message bits (the $x^r M(x)$ term) and r check bits (the $R(x)$ term). Furthermore, $W(x)$ is exactly divisible by $G(x)$. The division by

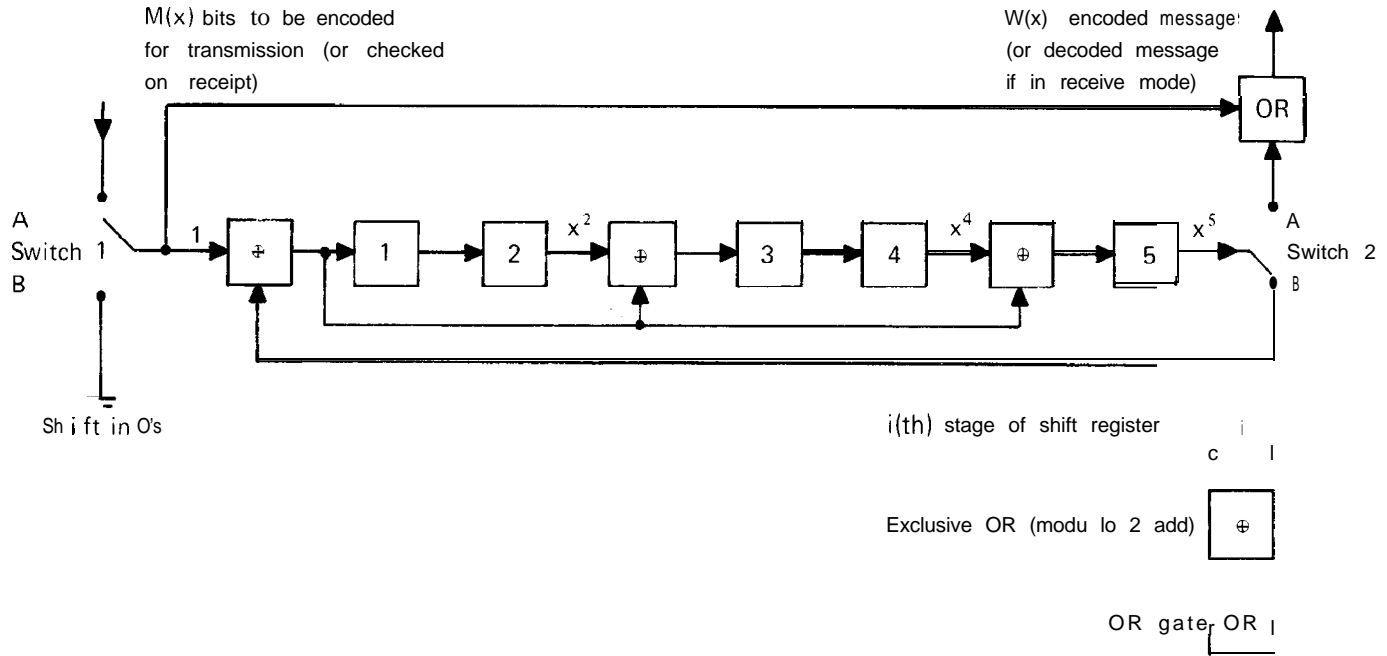


Fig. 2. Shift register for $G(x) = 1 + x^2 + x^4 + x^5$. Initially, the register contains 00000, switch 1 is in position A, and switch 2 is in position B. When all message bits have been transmitted, the register contains $R(x)$. Switch 1 now goes to B, and switch 2 goes to A to enable $R(x)$ to be shifted out. When data is being received, the resulting $R(x)$ must be zero; otherwise, the data is in error.

$G(x)$ at the transmitting end is accomplished by an r stage-shift register with feedback paths represented by the coefficients of $G(x)$, as shown in Fig. 2. On the receiving end, $W(x)$ is also divided by $G(x)$, and the remainder in this case must be 0; otherwise, an error has occurred.

Consider the following example related to the shift register shown in Fig. 2. Let the message be 1010010001. Therefore, $M(x) = 1 + x^2 + x^5 + x^9$. With $G(x) = 1 + x^2 + x^4 + x^5$, modulo 2 division of $x^5M(x)$ by $G(x)$ yields

$$Q(x) = 1 + x + x^2 + x^3 + x^7 + x^8 + x^9$$

and $R(x) = 1 + x$. Thus

$$W(x) = 1 + x + x^5 + x^7 + x^{10} + x^{14},$$

and the transmitted message is

11000		1010010001
CRC		original
bits		message
		bits

The remainder, $R(x)$, is generated by the shift register (which is initially at 00000) as follows:

Message Bit	Shift Register Contents
	Stage 12345
	10101
0	11111
0	11010
0	01101
	00110
0	00011
0	10100
	11111
0	11010
	11000

The final content is $R(x)$. Each successive shift register content represents a successive stage of the division of $x^5M(x)$ by $G(x)$, remembering that only the bits of $x^5M(x)$, which have been already transmitted at each stage, take part in the division. When all message bits have been transmitted, the contents of the shift register are shifted out by five successive right shifts to transmit $R(x)$. Note that during this operation, the zeros are shifted into stage 1 so that after $R(x)$ is transmitted, the contents are 00000; hence, the register is automatically cleared for more transmission.

Codes developed as described above are called "cyclic" codes. Such codes are used for error detection and correction for magnetic tape, disk, and data

communication. The generator polynomial $x^{16} + x^{15} + x^2 + 1$, for example, is widely used in synchronous data communication systems. It can detect all odd numbers of error bits, all possible single-error bursts not exceeding 16 bits, 99.9969% of all possible single bursts 17 bits long, and 99.9984% of all possible longer bursts. This is much better than simple parity checking, for instance, which detects only all odd numbers of error bits and no others. Note that parity checking is equivalent to having a generator polynomial $G(x) = x + 1$.

The study of cyclic codes revolves principally upon determining the code characteristics resulting from various generator polynomials. Peterson and Weldon (1972) and Tang and Chien (1969) give some applications and a more thorough mathematical treatment of cyclic and other codes.

REFERENCES

1969. Tang, D. T., and R. T. Chien. "Coding for Error Control" *IBM Systems Journal*, Vol. 8, No. 1, pp. 48-86.
1972. Peterson, W. W., and E. J. Weldon. *Error-Correcting Codes*, 2d ed. Cambridge, Mass.: M.I.T. Press.

J. S. SOBOLEWSKI

CYLINDER

For articles on related subjects see **MEMORY**: Auxiliary; and **VOLUME**.

Many rotating storage devices—drums, disks, data cells, and the like—have fewer read/write heads than recording tracks. Therefore, either the surfaces of these devices must move to position the desired information under a read/write head, or the read/write heads must move to hover above the appropriate recording tracks. The latter strategy is commonly used for large direct-access devices such as disks containing at least 20 million bytes.

For engineering convenience and efficient sequential processing of data, the following design has been adopted by most manufacturers of moving-head disk drives:

1. Tracks are numbered from top to bottom for

CYLINDER

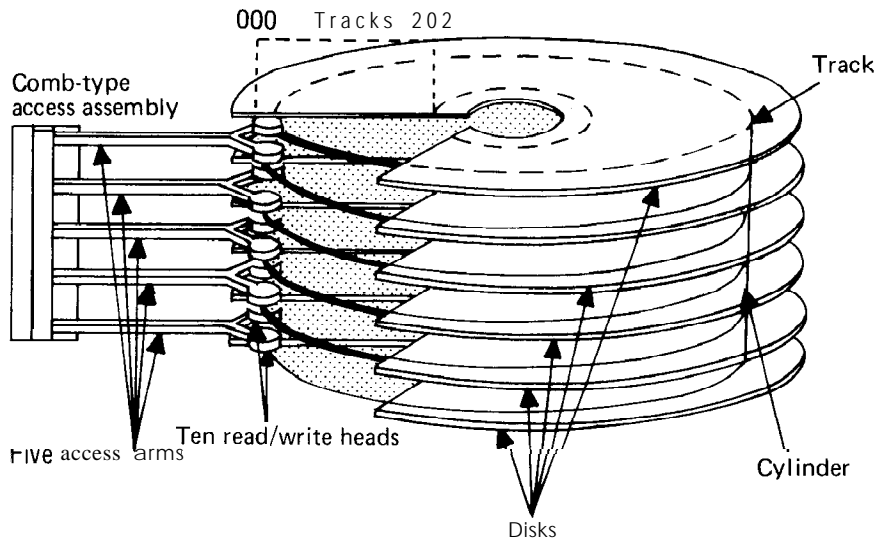


Fig. 1. Typical moving-head disk drive and pack.

each horizontal position of the read/write comb, as shown in Fig. 1.

2. During sequential writing operations, as the top track in each vertical plane becomes filled, control circuitry and system software allocate subsequent records to the beginning of the next track. When this is filled, records are started on the third track, etc.

3. Therefore, during sequential reading, a maximum amount of data can be read at one time before the comb must be moved. This is considerably faster

than the alternative strategy of writing all tracks concentrically on one surface before advancing to the next surface.

Each vertical set of tracks, one track per recording surface, is called a "cylinder," after the geometrical surface obviously outlined. There are as many cylinders per disk pack as tracks per recording surface (203 cylinders for the illustrated disk pack).

D. N. FREEMAN

DPMA. See **DATA PROCESSING MANAGEMENT ASSOCIATION.**

DANGLING ELSE

For article on related subject see **PROCEDURE-ORIENTED LANGUAGES.**

This picturesque phrase describes a situation in compound conditional statements where it may not be clear to which part of the statement an **ELSE** belongs. Consider, for example, the following statement in a PL/I-like syntax:

IFSEX = FEMALE THEN IF AGE \geq 30 THEN
RECONSIDER ELSE PURSUE (1)

If the **ELSE** belongs with the second **IF-THEN** pair, the statement implies pursuit of females under 30. But, if the **ELSE** belongs with the first **IF-THEN** pair, the statement implies pursuit of all males, but reconsideration of females 30 and over (and no information on what to do about females under 30; this would have to be taken care of in the next statement in the program, which would be executed if the first condition (**SEX = FEMALE**) were true and the second (**AGE \geq 30**) were false). Because of the ambiguity of which interpretation is correct, the **ELSE** is said to *dangle*.

One way of avoiding such ambiguities **is** by the use of compound statements. For example,

IF SEX = FEMALE THEN BEGIN IF AGE \geq 30 THEN
RECONSIDER ELSE PURSUE END (2)

makes it clear through the use of **BEGIN** and **END** that the first interpretation above is the intended **one**. (For another solution to the dangling **ELSE** problem, see Abrahams, 1966.)

In actual practice, of course, statement (1) must be given a specific interpretation by the compiler for the language [in **PL/I** the interpretation is that given in (2)], but writing statements containing a dangling **ELSE** is bad programming practice.

REFERENCE

1966. Abrahams, Paul W. "A Final Solution to the Dangling **ELSE** of Algol 60 and Related Languages," *Communications of the ACM*, Vol. 9, pp. 679-680.

A. RALSTON

DATA ACQUISITION COMPUTER

For articles on related subjects see **DIGITAL-TO-ANALOG CONVERTERS;** and **SPECIAL-PURPOSE COMPUTERS.**

DATA BANK

Computers have been used for decades to acquire and analyze data generated by instruments such as voltmeters, thermocouples, and electro-mechanical relays in factories, refineries, missiles, or aircraft. Typically data acquisition computers have fast memory-cycle times, so that bursts of signals from real-time physical processes like video scan devices will not be lost. Word sizes for data acquisition computers are short: 16 to 24 bits. Floating-point instructions are generally unnecessary, since data is inherently scaled within ranges determined by the processes being measured.

The main components of a data-acquisition computer are as follows:

1. Analog and digital input cables.
2. Analog to digital converter.
3. Disk or tape cassette for data storage.
4. Central processor.
5. Main memory.
6. Operator console.

For low-volume data acquisition, a paper-tape punch may be substituted for the disk drive or tape cassette.

Programs are loaded from punched paper tape, punched cards, or-in some newer models-from a host computer over a communications link.

Prices of data acquisition computers have decreased considerably since 1965, improving their advantage over manual methods for capturing and transcribing data in many applications. Their inherent reliability-especially central processor, main memory, and disk/tape components-has risen to such levels that they may operate unattended for days at a time.

Many data acquisition computers have been "ruggedized" to function in high-temperature environments such as steel plants or high-acceleration environments such as spacecraft. Data acquisition computers can be built and enclosed in small cabinets, some no larger than a desk or even a briefcase.

For the future, small low-cost data acquisition computers will be installed increasingly close to points at which original data is generated: factory floors, cash registers, continuous-process plants, etc. They will be connected by medium-speed (2,400 bits per second) telephone links to large central computers, which will periodically poll them for data acquired since last polling, status reports on processes being supervised, and hardware-reliability reports on the data acquisition computers themselves.

D. N. Freeman

DATA BANK

For articles on related subjects see **COMPUTERS AND SOCIETY**; **DATA BASE AND DATA BASE MANAGEMENT**; and **FILES**.

A data bank is a file of data derived from a variety of sources and stored in a manner suitable for ready access by a number of users. The term usually denotes a computerized file of personal data about identifiable individuals, which is maintained by an organization that may be able to affect some aspect of personal life and which may be used in making decisions about persons. The term first came into use in the late 1960s for a data system in which a municipal government could combine school, tax, utility, welfare, health, police, and other files. All departments of government could then draw data from this bank in making planning and operational decisions.

Early versions of such municipal management systems encountered strong public protest that the existence of such systems would be a serious threat to personal privacy, since the essence of privacy is that a person should be able to exercise reasonable control over the circulation of information about himself. After investigating the operation of actual data banks in a study for the Computer Science and Engineering Board of the National Academy of Sciences, Westin and Baker reported (1972) that computerized data banks typically contain about the same information as the manually maintained files they replace and are used in much the same ways. Furthermore, they found that computerization of working files had not affected the interchange of information among organizations to any great extent, and had specifically not resulted in the creation of networks of computers through which individual dossiers could be assembled.

Westin and Baker warned, however, that future technical developments might change the situation, and recommended that the federal government set up comprehensive legal protection for the privacy of computerized personal data. However, at the time of their recommendation, Congressional hearings on data banks had already been held, and the Department of Health, Education, and Welfare (under Secretary Elliot L. Richardson) had formed the Secretary's Advisory Committee on Automated Personal Data Systems to prepare the guidelines for protection of the many data banks operated by HEW. The Committee's report recommended that the following principles be made the basis of a code

of fair information practice, with statutory penalties to be provided for failure to observe the code in the operation of a data bank:

There must be no personal data record-keeping systems whose very existence is secret.

There must be a way for an individual to find out what information about him is in a record and how it is used.

There must be a way for an individual to prevent information about him, which was obtained for one purpose, from being used or made available for other purposes without his consent.

There must be a way for an individual to correct or amend a record of identifiable information about him.

Any organization creating, maintaining, using, or disseminating records of identifiable personal data must assure the reliability of the data for their intended use and must take precautions to prevent misuse of the data.

The issue of data banks versus privacy has arisen in most of the advanced nations of the world. Nearly all of these nations have taken steps to develop legal structures to allow the evolution of efficient technical methods of dealing with the large masses of personal data required in the running of a modern state while at the same time preserving the fundamental liberties of citizens.

REFERENCES

1967. Westin, Alan F. *Privacy and Freedom*. New York: Atheneum.
1971. Westin, Alan F. *Information Technology in a Democracy*. Cambridge: Harvard University Press.
1972. Miller, Arthur R. *The Assault on Privacy: Computers, Data Banks, and Dossiers*. Ann Arbor: University of Michigan Press (New York: Signet Books).
1972. Westin, Alan F., and Michael A. Baker. *Data Banks in a Free Society*. New York: Quadrangle Books.
1973. U.S. Dept. of Health, Education, and Welfare; Secretary's Advisory Committee on Automated Data Systems. "Records, Computers, and the Rights of Citizens." Cambridge, Mass.: M.I.T. Press (Washington, D.C.: U.S. Government Printing Office).

D. H. LUFKIN

DATA BASE AND DATA BASE MANAGEMENT

For articles on related subjects see **DATA BANK; DATA SET; FILES; and PROGRAMMING LANGUAGES.**

The term "data base" has yet to achieve a widely accepted standard meaning. However, it is to some extent accepted as conveying a more sophisticated concept than the older term "file," which was carried over into data processing terminology from the precomputer era. Unfortunately, it is all too frequently used when all that is implied is a conventional file. The difference between a data base and a file, in terms used prior to the advent of data processing, is perhaps analogous to the difference between a thoroughly cross-referenced set of files in cabinets in a library or in an office and a single file in one cabinet which is not cross-referenced in any way.

The important difference is that the *data base* must be stored in the computer on direct-access storage (such as disks) in order for the computer's central processing unit to be able to utilize the cross-references within a reasonable time. By contrast, a set of cross-referenced *files* could be theoretically stored on magnetic tape. However, the computer would then spend unacceptable amounts of time searching the tapes because it is not possible to access a specific data record on tape without passing over all other data preceding it on the tape. However, despite this disadvantage, magnetic tape is likely to remain the principal storage medium for archival computer files for many years to come, in view of its relatively low cost and high retention qualities.

The term "cross-reference" is not usually used when talking about a data base, the most usual term being "relation." One speaks of a relationship existing between types of records in a data base. A record type is analogous to a color-coded folder in a filing cabinet where different record types are segregated by varied colors. An individual folder may contain a reference to one or more other individual folders elsewhere in a set of cabinets. A referenced folder may have the same or a different color code as the folder that references it. In the data base, such relationships are stored in such a way that searching for records can be done directly without extensive cross-checking. Thus, the user has considerably more flexibility in the way in which he processes the data.

DATA BASE AND DATA BASE MANAGEMENT

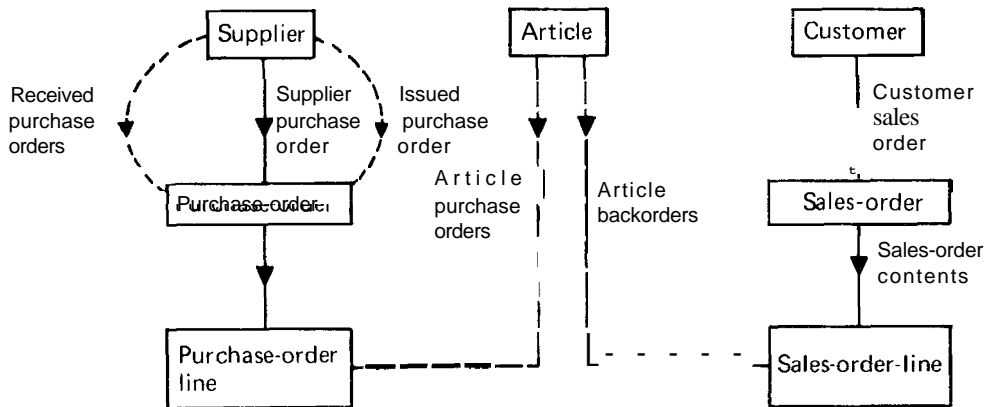


Fig. 1. Example: Components of data base in a warehouse system.

Example of a Data Base. As an example of a data base, it is interesting to consider the data processing in a warehouse (see Fig. 1). This example is a simplified version of one originally developed by Philips-Electrologica.

The warehouse needs to maintain data about each article to be stored. Articles are ordered in varying quantities by customers. If the stock of a particular article drops below a certain level, then new stocks are purchased from a supplier.

In the data base, it is necessary to have three principal record types: **ARTICLE**, **SUPPLIER**, and **CUSTOMER**. Each time a customer places an order for one or more different kinds of articles, a sales order is built up in the data base. In data processing terms, this means that there is also a **SALES-ORDER** record type. A single sales order may contain an order for several units of one article. More likely, a customer may order several different classes of articles on the same order. To facilitate the processing of the data base, **SALES-ORDER-LINE** is then regarded as a different record type, i.e., different from **SALES-ORDER**, which contains data about the whole sales order, customer identification, date of issue, and so on. The other record type contains data about each line entry in the sales order, such as which **article** the line refers to and how many units are ordered.

The situation is similar for the purchase order that the warehouse issues to a supplier when the stock of an article is discovered to be low? but there is a slight difference. A sales order and all its lines enter the data base together. A purchase order to a supplier is built up during the course of a day's or a week's processing. At a certain time, a program is run to issue the purchase orders. How many lines there are in a purchase order depends on customer

demands for different articles that a supplier provides and also on how many articles the warehouse needs to replenish stock and restore it to its normal inventory level.

In summary, the data base contains seven different *record types*: the three principal record types (article, supplier, customer) and the four respective subsidiary records of sales order, sales-order-line, purchase order, and purchase-order-line.

More important from a data base point of view are the various *relationships* that are defined among these record types. In Fig. 1, eight relationships are illustrated. Four of these eight are denoted by continuous lines, and four by dotted lines. The former are referred to as automatic and the latter as manual relationships. There is an important distinction between these two classes, which is best illustrated by an example.

The relationship between a customer and his sales order is automatic. There is no sense in having a sales order record in the data base if it is not clearly associated with a customer. Furthermore, it is unlikely that a need would arise to move a sales order from one customer to another. Normally, a sales order record would be entered in the data base and immediately associated with a customer record that is already there. In fact, if the sales order contains invalid customer identification, then the sales order should not be allowed in the data base at all. Hence, one can say that each time a sales order record is stored in the data base, it is "automatically" connected in some way to a customer record.

The relationship between a sales-order-line and an article is conceptually quite different. In Fig. 1 this relationship is called "article back orders." It is important to note that the **SALES-ORDER-LINE** record

type is subordinate to two relationships, an automatic one connecting it **to SALES-ORDER** and a manual one connecting it to **ARTICLE**. The latter relationship is manual because a sales-order-line is *not* automatically connected to the article being ordered at the time the sales-order-line record is stored in the data base. The name of the relationship, namely, "article back orders," gives the clue to the circumstances under which the connection would be made. At the time the sales-order-line is being processed, it may be discovered that there are insufficient units of the article in stock to meet the order. It is then necessary to hold the sales-order-line in some way so that when new stock enters the warehouse and stock levels are updated in the data base, the back orders that have not been successfully processed can then be easily found in the data base and processed. Hence, when a relationship exists between two record types such that a connection is conditional on circumstances that can be determined only at processing time, the relationship is called "manual."

This entire example illustrates a fairly simple data base structure with seven record types and eight relationships. Of the relationships, four are automatic and four are manual. The fact that three of the record types are subordinate in two or more relationships indicates a structure that is more advanced than that of an ordinary file.

Data Base Management. Many techniques have evolved for managing data bases in direct-access storage. Such techniques are used by programmers as they attempt to minimize processing time or storage space required, or sometimes to maximize flexibility. In earlier days, these techniques were used by the customer's application programmer.

During the past decade there has been a significant move by hardware and software manufacturers toward embedding these techniques into a component of systems software. This component is now quite widely referred to as a "data base management system," or DBMS.

DATA BASE MANAGEMENT SYSTEM, DBMS. A DBMS, as mentioned above, is a piece of software. Some vendors regard it as being part of the operating system; others build it in such a way that it is very much an optional extra for which the customer must pay if he decides to use it. It must be emphasized that the reason for identifying DBMS as a piece of software is to avoid any confusion that may arise with what might be called an "integrated manage-

ment information system," or IMIS. An IMIS *uses* a DBMS. An IMIS comprises human components, appropriate administrative procedures, and application programs, which collectively provide information to management and to others.

Many would argue that an IMIS is not possible without a DBMS. A management information system may be completely manual, but the degree to which one could achieve integration is likely to be modest. The meaning of "integration" in this concept is discussed more completely later in this article under "Integration of Separate Applications."

Relationship between Data Base and DBMS. A data base is a set of data stored in some special way in direct-access computer storage. A DBMS is the software that handles the storage and retrieval of the records in this data base. The DBMS cannot exist without a data base. The DBMS is the active partner and the data base is the passive one.

A number of requirements are frequently stipulated for a DBMS in addition to the basic one of handling the data in the data base. Integrity, privacy, and data independence are those most frequently cited,

Integrity refers to the ability of the DBMS to protect the data base from hardware and software malfunctions. It should be possible to recognize a problem, report it, reconstruct the damaged part of the data base, and restart the processing. Collectively, these processes are often referred to as a "recovery" system.

Privacy identifies a capability to protect the data base against unauthorized access or modification. While the need for this kind of facility varies from one enterprise to another, it always becomes more critical if the data base is to be accessed from on-line terminals.

Finally, *data independence* is a capability that many users regard as of paramount importance. It is defined as the independence of the application programs to structural changes in the data base. In the example of the warehouse, it might be required to add one or two new record types and a new relationship. If the programs that act on the data base do not need to be modified, and possibly not even recompiled after the changes to the data base structure, then it can be said that there is a degree of independence. This capability serves to minimize the reprocessing problem, which has caused major expense when conventional programming methods have been used. Data independence is irrevocably associated with a DBMS. However, astute users have certainly been able to achieve some data independence without using a DBMS.

DATA BASE AND DATA BASE MANAGEMENT

In summary, a data base management system is a piece of software that is used to manage a data base. A data base is a collection of cross-referenced records stored in a computer's direct-access storage. If the cross-references are not present, then it is safer to say that the data is stored as a file. At the present state-of-the-art in data processing, using files is much more common than using data bases.

Pieces of software are also available for managing files in computer storage. Such a piece of software is called either a "data management system" or, sometimes, a "file management system."

Data Management System. Considerable confusion has been caused by mixing the two terms "data management system" (DMS) and "data base management system" (DBMS). Some vendors, both hardware and software, refer to their DBMS as a DMS. Others provide both systems and regard them as separate but related items. Some use the term "file management system" to refer to one or the other. In view of the relatively widespread use of the older term, DMS, to identify an operating system's component for handling files of data that are not cross-referenced, it is recommended that this separation and distinction between DBMS and DMS be actively promulgated.

In the more prevalent situation, where the DMS is different and separate from the DBMS, the DMS is always an operating system component that may be little more than a disk input/output controller, namely, a piece of software to transfer blocks of data between disk and central storage. In this case, the DBMS will always make use of the DMS.

The important element that differentiates the DBMS and the DMS is the ability to define and make use of relationships (i.e., cross-references). If the data is organized into a data base, then a DMS at any one time typically can only process any one file in the data base. There are other differences between a DBMS and a DMS, but these are outside the scope of the present article.

A question that frequently arises concerns how easy it may be for a user to advance from using the DMS to using the DBMS, if and when a DBMS is available. Since using a DMS implies using the operating system facilities for input/output, and is also a fairly conventional approach to application design, a user's decision to use a DMS has no particular significance.

If a vendor does differentiate between his DMS and his DBMS, he may well underplay the problems of migrating from DMS use to DBMS use. Experienced users are justifiably cautious about the

problems they predict as a result of any major revolution in the way they do their data processing. They tend to request that any new DBMS they order should make the migration from their present conventional systems as easy as possible.

Vendors are equally aware of the problem of changeover and tend to stress how easy it is to move to the new system. While generalizations on this issue are necessarily vague, apparently it is easier to migrate to a DBMS of modest capabilities than it is to a more powerful one. It may also be possible to migrate from a DMS to a DBMS without actually taking advantage of the added power of the DBMS.

Classes of DBMS. In order to clarify the origin of the term DBMS, it is important to note that two basic classes of DBMS have been developed over the years. The two classes are now generally known as "host language DBMS" and "self-contained DBMS." It is the purpose of the following sections to describe and explain the difference between these two classes.

HOST LANGUAGE DBMS. A host language DBMS is simply one that, from a programmer's point of view, represents an extension to an existing programming language. The programming language is often Cobol, but PL/I and Fortran are also used. The extensions are new procedural statements that enable a programmer to access and modify the data records in the data base. This class of languages came into being as the capabilities of computers increased and applications became more sophisticated.

In early days when computers were first used, applications were recognized by users as being "suitable for computerization." What constituted an "application" was in those days rather effectively limited by two factors. One was the early dominance of magnetic tape as a main storage medium for the data. The second was the widespread use of Cobol as a language for programming data processing applications. The design of Cobol in 1959 was itself dominated by the method of processing sequential files stored on magnetic tape. Although some ad hoc modifications were made to the language during the mid-1960s, this basic design philosophy is still prevalent in Cobol today. A process has been under way for some years to update the whole approach.

As some sophisticated users gained experience and insight, and as direct-access storage became more readily available, they developed a somewhat larger view of what constituted an application. They soon realized that this broader scope of application would require better techniques for organizing (or

arranging) their application data in the computer direct-access storage. These users felt the need for cross-referencing their data records more thoroughly, using the approach discussed in the opening section of this article. They wanted to take advantage of the extra dimension offered by direct-access storage. More important, they wanted to provide better information to their management at less cost. They felt that this could be done only by "integrating" the data from the files of previously separate application programs into a centralized data base.

The technique of cross-referencing was initially employed by the users' programmers as they built "tailored" (or made to measure) software systems. As the cost of programmer talent increased during the 1960s, largely due to personnel shortage, this tailoring of each individual system became too costly. However, the manufacturers' software designers were fortunately learning how to generalize. In other words, they began to build generalized systems (now called DBMS), which allowed the user to make use of the cross-referencing technique without the expense of tailoring them to each software system.

At this point, the host language DBMS began to come into its own, often as an integral part of the operating system. It provided the more powerful structuring facilities that users needed to take a bigger view of their applications, namely, to integrate them. The DBMS made use of the more readily available direct-access storage in order to do this. Applications programmers then did not have to get involved with the details of physically accessing the data on the direct-access storage.

It must be emphasized that during the period between 1963 and 1969, these application techniques were used relatively infrequently. The concepts were not effectively promoted by those who did understand them, and many users had other, more crucial problems that took precedence over the one that a DBMS of this kind was intended to solve. Such other problems included operating system problems, compiler problems, and the design of the system using conventional techniques.

SELF-CONTAINED DBMS. A self-contained DBMS differs from a host language system in that it is in no way an extension of an existing programming language. On the contrary, it is usually quite independent of any language, although there are examples of self-contained DBMS that can process Cobol files. Recently, self-contained systems have been modified to provide a capability to act on the data structures processed by host language systems.

In this context, the self-contained DBMS becomes less self-contained and becomes essentially a supplementary facility in a host language system.

The reasons for developing the self-contained class of DBMS are quite different from those for the host language systems. Throughout the 1960s, and indeed continuing through the 1970s, there has been justifiable concern in many data processing circles about the increasing complexity (and associated cost) of programming, even when using higher-level languages such as Cobol. Consequently, efforts have been made to develop languages that are even easier to use than Cobol, and which therefore can be used by the "nonprogrammer." Another rationale motivating the development was the desire to have quick and easy access to information stored in conventional data files.

The effort to bring nearer to the man the interface between man and machine resulted in a plethora of these languages and associated software packages during the 1960s. Over the years a vast amount of resources has been directed toward this end and is still being expended in many research-oriented organizations.

The terms "retrieval language" and "query language" are often used when referring to "self-contained DBMS" because the major facility in any self-contained DBMS is indeed a query language. Others facilities include update and data definition.

RELATIONSHIP BETWEEN THE Two CLASSES. It is significant that, historically, there has been no coordination in approach or in thinking between the proponents of the "host language" approach and those advocating "self-contained" systems. On the contrary, the two groups have been in frequent disagreement as to which approach is more meaningful, but since 1971 there has been a growing recognition that the two approaches should be unified. Generally speaking, host language systems have become more important, simply because their very *raison d'être* has been to facilitate integration of smaller applications into a larger coordinated system so that several application programs can access a common or shared data base.

On the other hand, the self-contained system becomes really important to any user after a decision has been taken to utilize a host language system. The techniques embodied in self-contained systems are those that can help meet information needs identified by planning groups and by the higher echelons of management.

The trend of the current state-of-the-art is to make the two classes of systems gradually compatible. It is unreasonable to expect a user organization

DATA BASE AND DATA BASE MANAGEMENT

to store its data in two ways—one for the host language DBMS and the other for the self-contained DBMS, which until 1971 was invariably the case. Since that time, those concerned with software implementation have realized the problems created, and many are developing self-contained systems to operate on the data bases or files primarily created for processing using a host language system.

Integration of Separate Applications.

Having outlined the concepts of data base, DBMS, and DMS, it is appropriate to examine the task that the DBMS is intended to perform and the problem it is intended to alleviate in any user environment in which it is adopted.

The word "integration" has been used a number of times so far in this article. Experience has shown that, as with many other terms used in data processing terminology, "integration" is open to a number of interpretations. The purpose of the following paragraphs is to identify these and to emphasize the importance of deciding which of these is relevant in any given situation.

For the purpose of discussion, consider two application systems A and B. Whether A and B are major systems or merely components of larger systems is irrelevant to the present discussion. A and B may have some data that is common to both. This means that at least one item of data is updated and used by A, and is also required by B for inclusion in one of its reports or for use in one of its calculations. There are several ways in which system B can get access to system A data.

System A generates reports. Those generated may include some that are specifically designed to meet the requirements of system B; in other words, these reports are requested by the people responsible for system B. If system B indeed requires some of A's data for its own processing, then it may be necessary to copy the data off system A reports onto system B input forms for repunching because system B requires only part of the data generated by system A or because system B requires it in a rather different form. If reprocessing of the data is necessary for any reason, then it may be asserted that system A and system B are *not integrated in any sense*; in other words, they are totally uncoordinated.

Suppose, for example, that system B may require system A data, but *in a different form*. This normally means that some item of data that is common (at least in concept) to both systems has two sets of values, one for system A, one (maybe partially overlapping) for system B. On the other hand, the set of values may be the same for each system, but the

way they are represented to the computerized system may differ. This introduces the topic of *data representation standards*, which is a problem that must always be addressed if a satisfactory degree of integration is to be achieved. It may happen that system B requires the data in exactly the same form as it is generated and processed in system A. This in itself represents a modicum of integration, by the very necessary standardization of data representation. No meaningful integration of any kind is conceivable unless there is this standardization of data representation.

The designers concerned with systems A and B may agree to pass data from A to B in mechanized form rather than by passing reports from which the system B people can then punch what they need. This level of integration can be thought of as *mechanized interaction*. In this case, system A would arrange to generate files (usually on tape) and system B would include input programs to read these files. This situation represents the first meaningful level of integration. Although magnetic tape files are suggested here, it is possible that the transmittal media could be some other device, such as movable disks. It is implied that system A and system B are separate systems and that they are run at separate times, possibly on separate machines at the same or separate geographic locations.

The most complete degree of integration possible would be achieved if system A and system B were to run concurrently on the same machine. The data they could process would be in one of the following three classes: local to A, local to B, or common. Because the data common to both is the most important consideration, the designers responsible for the two systems must agree on record types to contain the common data. The programs of both systems may be written to access these record types, normally with agreement that only one system—for example, system A—is allowed to update any data in the record, while system B processing is limited to retrieval. It can then be said that systems A and B are *integrated*.

In summary, the following spectrum of potential situations can be identified when two application systems use data conceptually similar:

1. *Totally uncoordinated*: Each system has its own representation and does its own data collection.
2. *Partially interacting*: One system repunches data (and in so doing modifies the form) from reports generated by the other system.
3. *Mechanized interaction*: One system generates files, which can be used as direct input to programs

in the other system. No modification of data representation is necessary,

4. *Integrated:* The two systems have agreed on record types to be used by both systems. Programs of both systems access the records as stored in direct-access storage. There is agreement on which system has authority to update the data contained in records of this type.

Role of DBMS. A host language DBMS offers a common and widely used approach toward achieving the maximum integration possible. A self-contained DBMS is of somewhat peripheral interest to this central problem of integrating systems, but nevertheless it has an important role to play in allowing a user to provide for simple ad hoc information requirements that arise in an enterprise.

REFERENCE

Philips-Electrologica BV, "An Application Example of the CODASYL DBTG Proposal," Publication No. 5 122 991 2415 1 . Eindhoven, The Netherlands.

T. W. OLLE

DATA COMMUNICATION NETWORKS

For articles on related subjects see **ARPA NETWORK; COMPUTER NETWORKS; DATA COMMUNICATIONS; MODEM; MULTIPLEXING;** and **TELEPROCESSING SYSTEMS.**

Data communication may be thought of as the transportation or transmission of data from one location to another location. This requires a data communication network. Such a network consists of a set of nodes connected by a set of links. The nodes may be computers, terminals, or some type of communication control units in various locations, while the links are the communication channels providing a data path between the nodes. These channels are usually private or switched lines leased from a common carrier. Since the transmission over these lines is generally analog, while the data at the nodes is digital, data sets or modems are used to provide the interface between node and link. A simple example of a network is shown in Fig. 1,

where the links between modems are the communication lines that may be leased from a common carrier. The communication control unit in city E is used to multiplex or concentrate several lower-speed terminals onto a high-speed line. The single multidrop line also connects several terminals to the host. It is usually routed through several locations, picking up terminals along the way.

Current computer applications of data communication vary widely in function, scope, and requirements. They may be divided into six basic system types: inquiry and response, data collection, data distribution, conversational, remote batch processing, and message-switching systems. The number of such systems is growing very rapidly and will continue to grow. In large systems of this kind, the costs directly related to data transmission are a very significant part of the overall cost; hence the planning and design of data communication networks is very important.

Network Planning. Planning and designing a data communication network can be exceedingly complex because of the wide variety of requirements and the large number of possible solutions. The objective is to satisfy all requirements at least cost. Factors to consider in planning include:

1. The type of teleprocessing system used.
2. Volume and distribution of data to be transmitted.
3. Access and response times required.
4. Number and geographical location of nodes.
5. Type of terminal or equipment at each node and its transmission speed.
6. Error rate that may be tolerated.
7. Need for future expansion.
8. Availability, reliability, and maintenance of the network.

Network Design. Given the network objectives, the design process includes the choice of terminals, line control procedures, modems, communication control units, common carrier facilities, and network configuration. This can be a very lengthy process, since the number of possible solutions is very large and the various influencing factors are interrelated.

For example, Fig. 2 shows three basic network configurations: the point to point, the multidrop, and distributed connection, as well as a star network, which consists of four point-to-point connections. In the distributed network shown, more than one path exists between nodes. Most present networks are

DATA COMMUNICATION NETWORKS

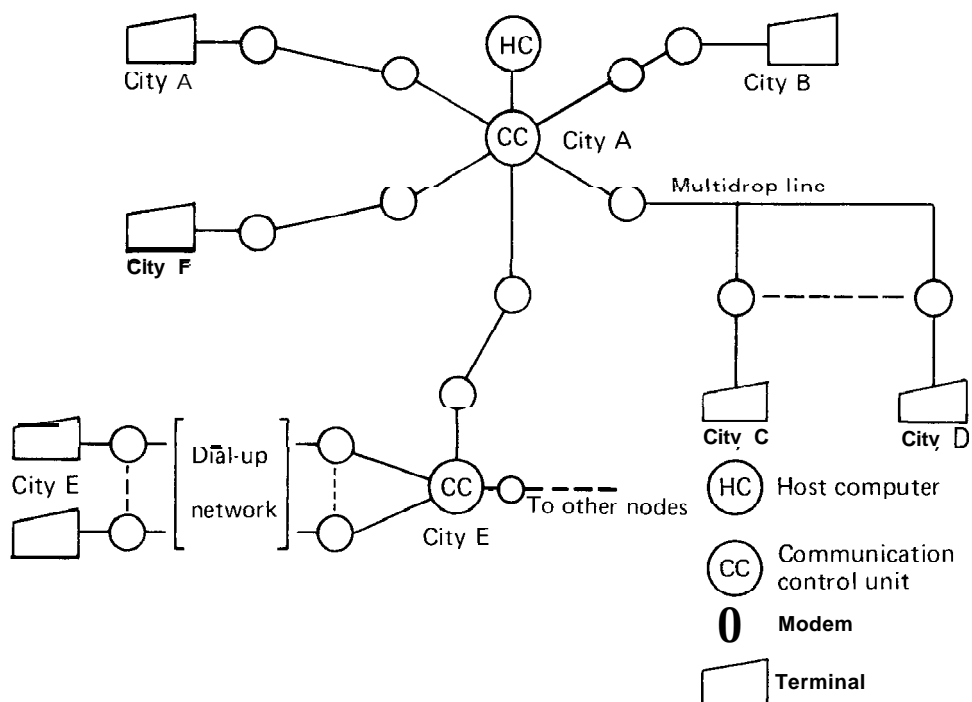


Fig. 1. Example of a data communication network.

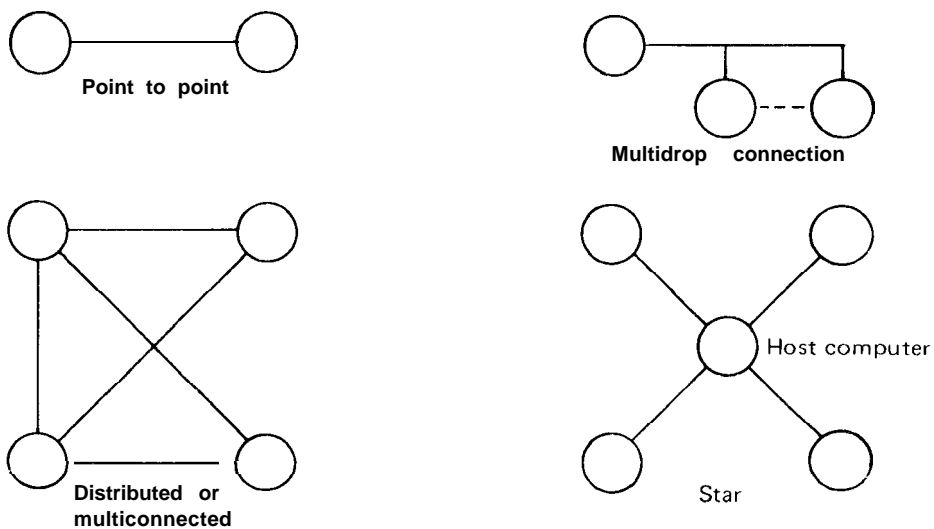


Fig. 2. Three basic methods of interconnecting nodes in a data communication network.

combinations of the three basic configurations.

The cost of the various types of lines or common carrier facilities required are governed by very complex tariffs based upon location, circuit length, and type of line. **Both** the geometry and types of lines can in turn be greatly influenced by using appropriate communication control units such as **multiplexers** or concentrators placed at strategic locations within the network. Such units can markedly improve the efficiency of the lines, reducing their number and hence the cost. Furthermore, performance-cost trade-offs are always possible.

Based on the above considerations, a great deal of experience is necessary to design an economical network. The design process is usually iterative in that the designer makes an initial guess at a possible network by deciding on the types of lines that should be used, the auxiliary equipment needed to improve line efficiency, and the network configuration. This initial design is tested to insure that it meets the planned objectives; if it does, it is then evaluated for cost. Special computer programs may then be used to find alternative approaches to minimize the network costs. These programs are usually modular, each module designed to solve a particular segment of the optimization problem, such as finding optimum locations of concentrators. Since computers cannot replace experience and intuition, the programs are interactive in the sense that the designer is in the feedback loop. Should the costs be too high, some of the planned objectives or specifications may be relaxed to lower the cost, and the process is repeated until an acceptable cost-performance network is found.

REFERENCE

1973. Martin, J. *Systems Analysis for Data Transmission*. Englewood Cliffs, N.J.: Prentice-Hall

J. S. SOBOLEWSKI

DATA COMMUNICATIONS

GENERAL PRINCIPLES

For articles on related subjects see **COMMUNICATIONS AND COMPUTERS**; **COMMUNICATION CONTROL UNIT**; **ERROR-CORRECTION**

ING CODE; **MULTIPLEXING**; **TELEPROCESSING SYSTEMS**; and **TERMINALS**.

For articles on related terms see **BAUD**; **CYCLIC REDUNDANCY CHECK**; **PACKET SWITCHING**; and **PARITY**.

From the first time that data had to be passed between one register and another in a computer, the problem of data communications had to be addressed. This article is concerned with the transmission of data from its source to its destination, as shown in Fig. 1. This subject could cover the contents of many volumes. Three general textbooks (Davenport, 1971; Martin, 1969, 1970) are recommended for additional reading. Other references (Bullington et al., 1959; Kretzmer, 1969; Petersen, 1961; Shaw, 1969) deal with more specialized areas.

Normally (see Fig. 1), data is passed in parallel between a computer or peripheral in finite-sized chunks (e.g., 8-bit bytes) to a register, shown as *SO*. This data must be passed via a communication network (*CN*) to a sink (*SI*), where it is passed on in the same or different finite-sized chunks to another computer or peripheral. The communication network usually has the property that the part of it dedicated to the communication between *SO* and *SI* can carry only one bit at a time. Therefore, the data from the *SO* must be serialized in the parallel-series converter (*PS*) and deserialized again in the series-parallel converter (*SP*). The data output of *PS* is usually a bistable binary signal that can be interpreted as one of two states: 0 or 1. To pass through the *CN*, it must usually be used to modulate an analog signal or to emit a short pulse. This is achieved in the modulator (*MOD*), and the digital level is recovered in the demodulator (*DEM*).

If information can flow only from *SO* to *SI* (Fig. 1), the communication is said to be "simplex." If data can flow both from *SO* to *SI* and from *SI* to *SO* simultaneously, the communication is called "duplex." If the data flows in these two directions do not proceed simultaneously, the communication is said to be "half-duplex." In some cases, the communication channels themselves may be full duplex, but either the hardware of *SO* and *SI* or the software associated with them may restrict the communication to half-duplex. Since it is usual for the communication portion of the circuit to be at least half-duplex, it is normal for the functions of modulator and demodulator (Fig. 1) to be combined. The resulting equipment is called a "modem."

Often a character input from a terminal to a computer will be echoed back onto the terminal's printer to show it was received correctly; this mode

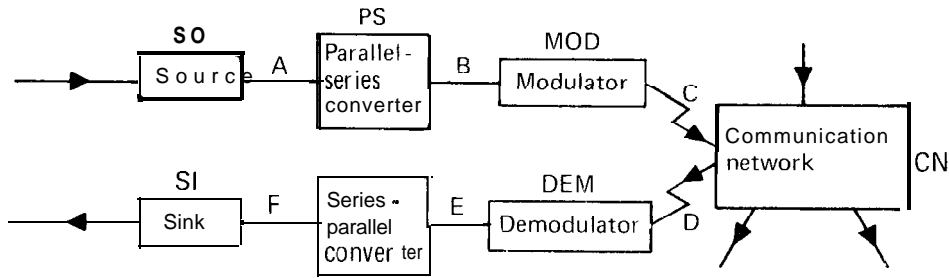


Fig. 1. Schematic of communications between source and destination.

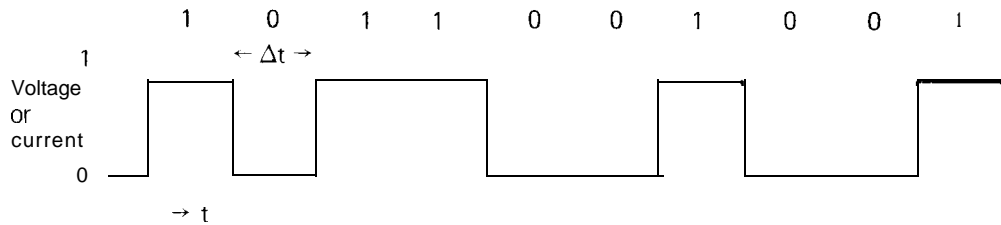


Fig. 2. Example of telegraph modulation.

of working is called an "echo-plex." The devices *SP* and *PS*, and the interfaces *A* and *F*, are also often combined, with additional buffers in each to permit duplex working.

It is beyond the scope of this article to say much about techniques of modulation (see Davenport, 1971; Martin, 1969). The simplest way to modulate signals is to use telegraph techniques to insure that the channel has one of two states: with current or without current. An example of this form of signaling is shown in Fig. 2.

The fastest signaling rate of a communication channel is called the "baud rate." In the system shown in Fig. 2, the baud rate is $1/\Delta t$. When only two-level signaling is used, the baud is also equal to the rate of information transfer in bits per second (bps). If multiple-level signaling is used, as shown in Fig. 3 for four-level coding, then the bit rate is higher than the baud rate. To obtain the signals in Fig. 3, each pair of bits in Fig. 2 is taken together, and the four resulting combinations (00, 01, 10, 11) are each coded to one level. Clearly, this approach can be extended to n levels, but the circuitry required to discriminate and decode the levels becomes increasingly complex.

The form of signaling described above has problems in long-distance transmission, and it is more usual to use the pulsed signals shown in Fig. 2 or Fig. 3 to modulate the amplitude, frequency, or

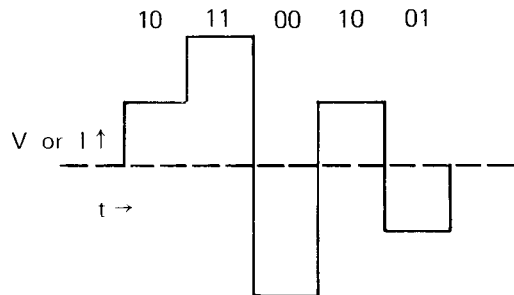


Fig. 3. Example of four-level coding.

phase of a carrier sine wave. These forms of transmission are called "analog transmission," and are well suited for use over the conventional telephone system, which is designed for the transfer of analog signals.

Each single communication channel has a certain *bandwidth*. For example, the amplitude response of a standard telephone channel, sketched in Fig. 4, has a *bandwidth* (within a frequency range outside of which the transmitted attenuation rises rapidly) of about 3 kHz (the characteristics are good from about 300 Hz to 3.3 kHz). It has been shown by Nyquist that for all methods of modulation, the maximum signaling rate is about twice the frequency bandwidth. Thus, for a 3 kHz telephone channel, the

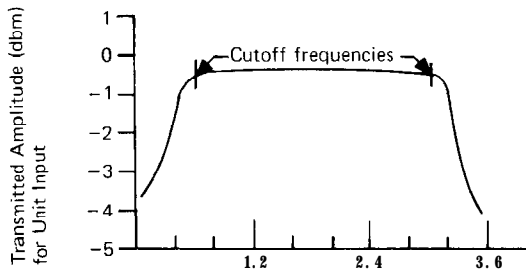


Fig. 4. Frequency (KHz)

maximum signaling, or baud, rate is about 6K baud. The maximum information transfer rate is related to both the baud rate and the number of levels of coding used. Claude Shannon has shown that the maximum information transfer rate, or channel capacity C , in a noisy channel is

$$C = BW \log_2(1 + S/N)$$

where BW is the bandwidth and S/N is the ratio of signal strength to random noise level, called the "signal-to-noise ratio." Thus, the more noisy the channel, the less can multiple levels be tolerated. If S/N is 15, the preceding equation shows that the channel capacity is $BW \log_2 16$ (i.e., $4 BW$) so that four-level coding could be used. With present telephone channels, the maximum signaling rates used are 3.2K baud, with eight-level coding (3 bits) to give 9.6K bps.

It should be noted that, just as two amplitude levels were used in the telegraph modulation of Fig. 2, two frequencies or two phases could be used for frequency or phase modulation of a carrier line wave. For higher-level signaling, more frequencies or phases are used. Thus, for the system of Fig. 3, the pairs of bits, or "dibits," (00), (01), (10), (11) would each be made to refer to a discrete frequency or

phase with multilevel frequency or phase modulation. A good discussion of sophisticated multilevel modems is given by Kretzmer (1969).

The mode of modulation may make it possible for the modems to generate timing pulses themselves; such a system is called "synchronous." Both with phase-modulated analog signals and pulse-code modulation, it is possible to synchronize the two modems and produce timing pulses in the modem to indicate when a bit is being sent or received.

A simpler modem is possible when such synchronization is not required. Moreover certain man-oriented peripherals, such as keyboard terminals, need to send data only at irregular intervals. Such systems are termed "asynchronous"; in an asynchronous system, the signaling rate is predetermined, but it is necessary to indicate the start of each piece of information (usually a byte or a character) by sending a start-bit before, and one or more stop-bits (of opposite polarity) after transmission of the data. Thus, a byte 145 (in octal) would be sent as shown in Fig. 5. From the arrival time of the start-bit, the bit timings of the subsequent bits can be deduced. The stop-bit is required to insure for at least a one-bit time that the signal has an appropriate value by which a subsequent start-bit can be recognized.

It is even possible for the data format to be asynchronous (i.e., start- and stop-bits are included) with synchronous modems so that the transmission system is synchronous. Although the start- and stop-bits are redundant in such a system, it is often convenient to include them when the same electronics in PS and SO of Fig. 1 is to be used with different modems. As described before, an asynchronous system has a fixed length of byte, whereas a synchronous system may have a variable length (see below).

Recently, with the reduction in costs of digital circuitry, it is becoming possible to use pulse-code modulation for transmission. Here, each device is

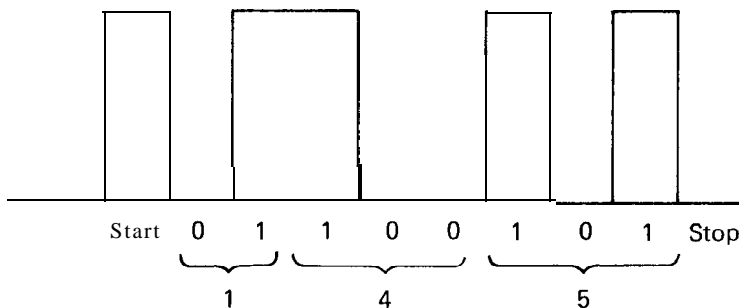


Fig. 5. Data sent for 145 in asynchronous system.

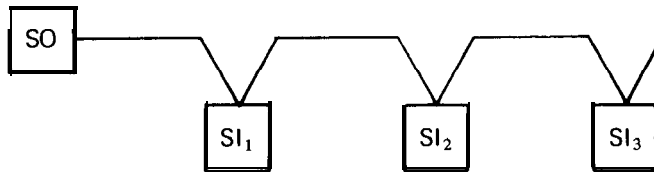


Fig. 6. Multipoint connection between SO and SI_1 , SI_2 , SI_3 .

given a time slot, and during this period either a pulse is put on the channel or not. This is a modification of the original telegraph techniques, and is the basis of the digital transmission being developed. This form of modulation is really two-level amplitude modulation in a synchronous system.

The communication shown in Fig. 1, in which two parties SO and SI are connected, is called "point to point." Alternate forms used in some applications are shown in Fig. 6. In the communication circuit depicted by Fig. 6, SO can send (or receive) data along the channel connecting it to SI_1 , SI_2 , and SI_3 . By appropriate signaling, it is possible to insure that the data is received at its correct destination, SI_i . This type of connection is called "multidrop" or "multipoint." In some cases it is desirable to have the same information received at all stations: SI_1 , SI_2 , SI_3 . This mode of communication is called "broadcast."

If several devices share the same communication channel, as shown in Fig. 6, conflict for use of the channel can occur. One mode of overcoming the conflict is to allow any device to request the channel at will; its efforts to put information onto the channel will then be detected by the others, who will refrain from putting on their information until the message-sending transmitter has ceased. This mode of using a channel is called "contention"; it works well on a point-to-point basis, but reasonably complex strategies must be adopted for successful contention on multipoint channels because of the perceptible delay between information being placed on the channel and its receipt by the other parties.

Another way of resolving conflicts is particularly useful if one device, shown as SO in Fig. 6, can be used to control the others. This mode of control is called a "polling" philosophy; in this mode of communication, SO will ask each SI_i in turn whether it has anything to send, or will address an SI_i if it wishes to send data to that device. Clearly, the polling strategy can be carried further; SO can poll one device SI_i and address another one, SI_j , to insure that the data is sent from SI_i to SI_j . Alternatively, it can poll to see if any device has data to

send. The whole question of address control is complex and depends on the nature of the communication channel. A normal telephone channel, for example, is usually point-to-point. A satellite communication channel is fundamentally broadcast, even if it is often used in a point-to-point manner.

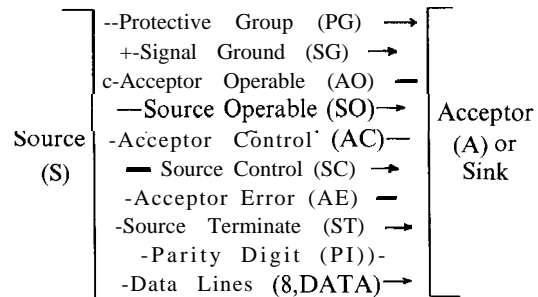


Fig. 7. British standard interface (simplified).

To illustrate the problems of the control and synchronization required between SO and SI_j , we consider the interface inside a single computer system. Here, the data communication path usually has a fairly complex hardware interface with lines for passing data and control. The sort of information passed is indicated in Fig. 7, which is a simplified version of the British standard interface. This interface is for a synchronous autonomous simplex transfer, with 8-bit data lines and a parity line (PD). The AO and SO lines in Fig. 7 are to assure each device that the other is operational. The AC informs the source when new data is required, and SC then informs the acceptor when the data is ready. If an error (e.g., parity) is detected, AE informs the source, and completion of block transfer is indicated by ST . The lines PG and SG are not really relevant. The former is the ground for the power connection and the latter is the ground for the signal lines.

The discussion of the interface of Fig. 7 illustrates one of the key features of data communication. In a local connection, there are a number of

control lines to establish synchronization, timing, acknowledgment, error detection, end of transmission, etc. All such control information in the data communication system of Fig. 1 *must be carried with the data*. Moreover, in a local system, errors in transmitting data over the interface of Fig. 7 are usually rare; over long distances, noise and other phenomena will often cause bits to be lost. Since some of these bits may contain control information, care must be taken in the communication environment to insure that the correct action will be taken in all cases at *both sides of the link*. We will discuss below how some of this control information is passed.

In the communication of Fig. 7, the data is carried across the interface in parallel; in that of Fig. 1, it must first be serialized. We discussed previously that in some systems the modems of Fig. 1 established synchronization with each other, the so-called synchronous systems in which the bit timings are developed in the modems. It is merely necessary to establish this synchronization at the beginning of the transmission. Since this takes some time, it is usual to send data in a synchronous system in a block with some header information, followed by the data, and with some control and error detection data at the end. A synchronous system can be used only if the source *SO* of Fig. 6 has a buffer so that it can collect a whole block of information before transmission begins.

We mentioned earlier that the data format could be asynchronous with synchronous data transmission; redundant start- and stop-bits would be transmitted. In the same way even an asynchronous communication system can be used to send block-oriented data, by prefacing it with an appropriate header and ending the block with appropriate end-of-block characters.

It is usual in a data communication system to send some bits additional to the actual useful data to identify the existence of, and possibly to correct, errors. The simplest error detection code is to add to each n bits of data an $(n + 1)$ st bit, so chosen that the sum of the $(n + 1)$ bits is of a given parity (even or odd); such an extra bit is called a "parity" bit. In the asynchronous transmission system of Fig. 5, such a parity bit is often sent immediately before the stop-bit.

While this code is simple, it is not adequate if high information integrity is desired. Noise in transmission lines occurs fairly often; 10^{-3} is a normal error rate on a switched line. Moreover, the nature of these errors is such that the noise that causes them often lasts more than one-bit time. For this reason,

most data transmission systems, other than those involving the simplest keyboard terminals, send their information in blocks and use more sophisticated error-detection codes that act on the whole block. One simple method considers the block as made up of n -bit bytes; it then does a parity check on the i th bits of each byte, and thus constructs the *i th bit of a blockparity check* byte. When this block-parity check is combined with a parity check on each byte, only errors that occur in rare combinations would remain undetected. A more sophisticated set of error detection codes is based on *cyclic redundancy checks*, which require rather more logic, but are even safer. The subject of error detection codes is discussed fully by Petersen (1961).

Just as a single byte in Fig. 5 of an asynchronous transmission system was framed by a start-bit (often a parity bit) and a stop-bit, so whole blocks are usually framed by some synchronizing bytes, a start-byte, error-detection bytes, and an end-of-block byte. In some cases the end-of-block byte is replaced by information in the header of the block, stating the number of bytes in the block. For the case of a multipoint or polling situation, the header may also contain polling or addressing information. An international standard on block structure has been developed (Shaw, 1969), but it is not yet in universal use. Most synchronous communication systems are synchronous at the bit level, but are asynchronous at the block level. For this reason, the header and the end-of-block bytes bear the same relation to the block as the start- and stop-bits do to the single byte shown in Fig. 5. Some special synchronizing bytes are sent in the header to obtain the bit synchronization achieved by the start-bit in an asynchronous system.

Just as the interface of Fig. 7 must have an error-return line, so it is usually necessary to acknowledge the correctness of each block sent. In some cases this acknowledgment is made before any new block can be sent. In others, a header contains a block number, which is increased each time a block is sent. It is assumed that each block has been received correctly, unless a *negative acknowledgment* is sent subsequently. If that occurs, either only the faulty block or all subsequent blocks also are retransmitted. The philosophy is particularly important when there are significant delays in the communication network (e.g., when one or more satellite hops are involved) requiring a minimum of 0.5 sec for a round-trip signal.

It is instructive to consider what speeds and modes of data communication are offered currently by the telecommunications authorities over the tel-

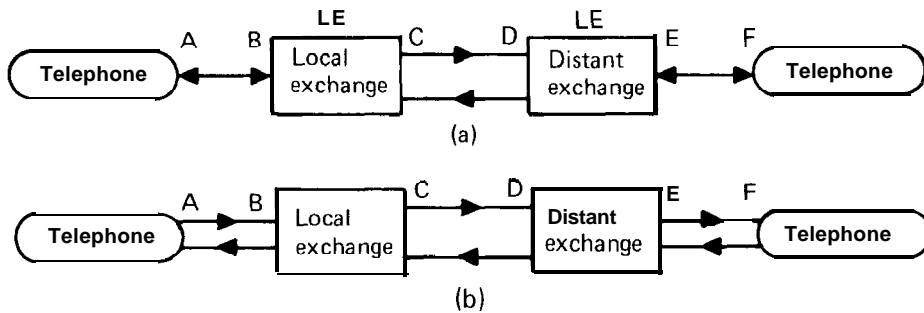


Fig. 8. Schematics of telephone networks. (a) Switched telephone line. (b) Four-wire leased telephone line.

ephone networks. They usually offer facilities over both switched and leased lines. In the former, it is possible to dial up any other subscriber on the switched network and to communicate with him; in the latter, connection can be made along only one path (possibly multidrop, as in Fig. 6). On a leased line, because only one path is used, it is possible for the telecommunications authorities to suitably *condition* the line to improve its performance; such conditioning is called **line equalization**. Alternately, both with switched and leased lines, it is possible to arrange for the modems to adjust to line conditions; this is called "equalizing" or "balancing" the modems. On a leased line, this balancing need not be done too often, unless very high performance is required, because the same path is always used. On switched lines, this balancing must be done on each call.

Fig. 8 illustrates the connection between two telephones and their local exchanges. Between the exchanges (C-D in Fig. 8), there are separate channels in the two directions, insuring duplex facilities. On a switched line, there is usually only one pair of lines, as shown in Fig. 8(a), between the telephone line and the local exchange; this is called a "two-wire circuit." On a leased line, it is possible to order at comparatively low cost a second pair of lines to the local exchange, as illustrated in Fig. 8(b). In this case, one has a four-wire circuit, and is able to operate at maximum speed simultaneously in both directions. On a single pair, as in A-B of Fig. 8(a), it is possible to work at fairly low speeds in both directions simultaneously. It is also possible to work at a much higher speed in one direction with a lower-speed return path. Schematically, this situation is then as shown in Fig. 8(b), but only one pair of physical connections need exist between A and B or E and F. This low-speed return path is called a "supervisory return," and varies in speed between 5

and 150 bps. It is used to turn around the line in the half-duplex situation or to signal acknowledgments or enter keyboard data in duplex. Thus, in the true four-wire case of Fig. 8(b), it is possible to have the high-speed data going simultaneously on each line, as shown in the figure, with additional reverse supervisory information. The range of facilities currently offered by the telecommunications authorities are illustrated in Fig. 9.

Finally, a brief discussion of multiplexing techniques is required. A very common situation in data processing involves heavy traffic between one cluster of terminals T_i and a distant computer C, as illustrated in Fig. 10. This is similar to the contention mode discussed in relation to Fig. 6. With the present tariff structure and traffic capacity of individual lines, as illustrated in Fig. 9, it is often more economic to use some form of concentrator (N), as shown in Fig. 10(b), to concentrate the terminal traffic and pass it over a leased connection to C. By adopting such a course, there will often be considerable saving in transmission costs over the solution of Fig. 10(a), which will more than compensate for the cost of the more sophisticated equipment at N. One reason for the lower cost is that direct connections to the T_j are often usable only up to 300 bps, whereas C-N can run up to 9.6K bps. Thus, by splitting up the channel C-N into a number of subchannels, the traffic can be carried more economically.

One method of achieving this subdivision is to allocate to each terminal T_j a frequency set $F_{j1}, F_{j2}, F_{j3}, F_{j4}$ (for both 0 and 1 in the two directions); this is called "frequency division multiplexing" (FDM). A second method is to allocate to each terminal a time slot on the channel; this is called "time division multiplexing" (TDM). These methods extend to the user portion of the system the advantage of cost-saving techniques presently employed by commu-

Speed Range (K bps)	Switched or Leased	Half- or Full Duplex	Line Equalization Required	Asynchronous or Synchronous	Levels of Coding	Note
up to 0.2	S or L	H	N O	A	2	Uses d-c telegraph techniques
up to 0.3	S or L	F	No	A	2	Uses modems
up to 1.2	S or L	H	N O	A	2	May have low speed return
1.2 to 3.6	s or L	H	N O	S	2-4	May have low speed return
Up to 4.8	L	F	No	S	4	Requires four-wire local connection for full duplex
up to 9.6	L	F	Yes	S	4-8	Requires four-wire local connections for full duplex
Up to 72	L	F	Yes	S	2-4	Requires special treatment of the line and possibly repeaters
Above 72	L	F	Yes	S	2-4	Required special local lines and transmission facilities
1544 or 2048	L	F	Yes	S	2	Standard pcm digital transmission

Fig. 9. Typical telecommunications facilities available.

nications carriers on their part of the network. The multiplexing techniques can be optimized, however, to the specific capacity, usage patterns, and geography of transmission and switching capacity anticipated by the user.

Still further saving is possible on transmission cost. The terminals T_j are often used only intermittently; a keyboard terminal capable of 30 char/sec is rarely run at an average rate higher than 3 char/sec. For this reason it is possible to store whole messages from or to T_j at the node N, and send them in one block over C-N. This form of line utilization is called "packet" switching. With frequency or time division multiplexing, the frequency or time of the bit identifies the terminal T_j ; with packet switching, this identification must be carried in a header.

Obviously, packet switching requires more storage and sophistication at N than the other techniques, and is justified only if the transmission costs

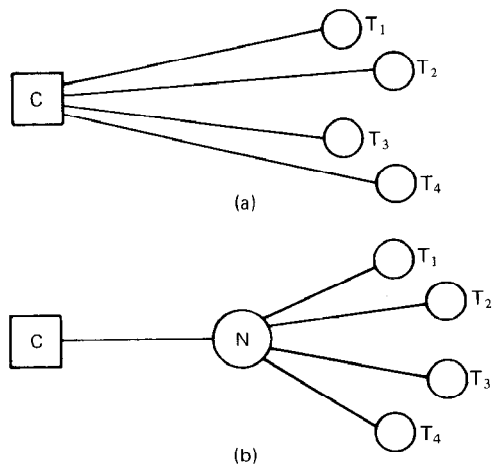


Fig. 10. Multiplexing. (a) Direct point-to-point connections. (b) Concentrated terminal usage.

DATA COMMUNICATIONS

are high. This is exactly analogous to the use of time-assigned speech interpolations (TASI) over transatlantic telephone lines to multiplex m telephone conversations over n communications lines, where $n \leq m$ (Bullington et al., 1959). By this method, messages are interspersed over a smaller number of channels whenever there is a lapse of voice communication on an active channel. When one message is interrupted, another message occupies the vacant time slot, and when the first transmission is resumed, the second message is shifted to the next vacant channel.

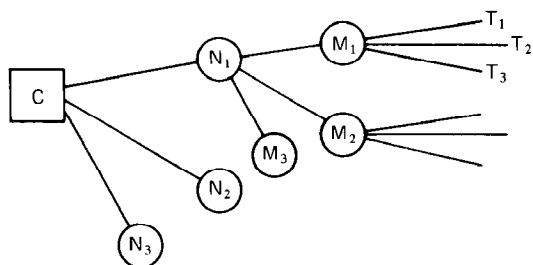


Fig. 11. Example of a concentrating hierarchy.

In real systems, a combination of FDM or TDM and packet switching is sometimes used. Over long distances, packet switching is used; over medium distances, FDM or TDM; and over short distances, point-to-point. This situation is illustrated in Fig. 11. Between C and N_j , where long distances may be involved, there may be packet switching (the N_j will have to be nodal computers); between N_j and M_i , there may be TDM or FDM transmission, while the terminals T_i will use the switched network (often using only local calls) to access the nearest M_i on a point-to-point basis. The question of reliability is too complex to be considered in a brief survey article. However, in passing, it may be mentioned that the M_i may be so located that a terminal may access more than one station, and that the route from M_i and N_j to C may lie via different N_j to give better access in case of failures of a single M_i , N_j , or the lines between them. More information on multiplexing techniques and the optimized design of communication systems is given by Davenport (1971), and Martin (1969, 1970).

REFERENCES

1959. Bullington, K., et al. "Engineering Aspects of TASI," BSTJ, Vol. 38, No. 2, pp. 353-364.
1961. Petersen, W. W. *Error Correcting Codes*. Cambridge, Mass.: The M.I.T. Press.
1969. Kretzmer, E. R. "Modem Techniques for Data Communication over Telephone Channels," *IFIP '68 Proceedings*. Amsterdam: North Holland Publishing Co., pp. 716-721.
1969. Martin, J. *Telecommunications and the Computer*. Englewood Cliffs, N.J.: Prentice Hall.
1969. Shaw, R. T. "Basic Control Procedures for Data Transmission," *IFIP '68 Proceedings*. Amsterdam: North Holland Publishing Co., pp. 728-733.
1970. Martin, J. *Teleprocessing Network Organization*. Englewood Cliffs, N.J.: Prentice Hall.
1971. Davenport, W. P. *Modern Data Communication*. London: Pitman Publishing.

P. T. KIRSTEIN

SOFTWARE

For articles on related subjects see **DATA COMMUNICATION NETWORKS**; **DATA COMMUNICATIONS**: General Principles; **ERROR-CORRECTING CODE**; and **SOFTWARE**. For articles on related terms see **BUFFER**; **INTERRUPT**; **REAL-TIME APPLICATIONS**; and **REGISTER**.

The transmission of messages by electrical means has been a practical reality since the middle of the nineteenth century; the basic concepts in use today do not vary greatly from those used by Samuel Morse in 1838. Morse transmitted his messages a character at a time; each character was encoded as a sequence of "dots" and "dashes." Character encoding and transmission was done by a human operating a key that controlled a buzzer at the receiving point. The reception and decoding was done by the human ear. When the message was received and transcribed on paper, it was either delivered locally or retransmitted to a remote destination. Transmission errors were detected by the human ear.

Some keyboard machines have been developed to both encode and decode the Morse code, but the code does not lend itself to simple mechanization because of the variable lengths of the individual character codes. However, with the introduction of constant character-length codes such as the Baudot code, the mechanization of the character encoding and decoding had become standard in almost all

data communication environments by 1950. The next step in data communication mechanization was the introduction of the computer, which made possible the automation of line control, error control, and message-routing procedures.

Data communications software functions that exist in time-sharing networks, military command and control, airline reservation, telegraph, and factory data collection systems are largely application independent. Variations in the content of communications software are caused by the differences in traffic rates, line disciplines, volumes, and geographical separations rather than by differences in application. In addition to providing the functions of message transmission, message reception, error control, and message routing in a communications environment, the data communications software provides an interface between the communications environment and the EDP environment.

Before discussing the above functions in more detail, we will examine from a software viewpoint the characteristics of the communication lines that connect together the terminals and computer to form a data communications network.

Communication Line Characteristics. Communication lines provide a single signal path from transmitter to receiver; information is sent across this path serially, bit by bit. Because there is only one serial path between the transmitter and the receiver, message control information has to be sent along the same path as the message. Over the years, several different message control techniques for distinguishing between the control and message information have developed. Stutzman (1972) gives an excellent review of the techniques used in this area.

The directional characteristics of the line are described by the terms "simplex" (one way only), "half-duplex" (two-way alternate), and "full-duplex" (two-way simultaneous). In a simplex path, information can be transferred only in one direction; e.g., a stock market "ticker tape". Half- and full-duplex lines are capable of transmitting information in both directions. A full-duplex line allows simultaneous transmission of information in both directions; a half-duplex line allows transmission of information in one direction at a time. On a half-duplex line, the line has to be "turned around" before it can be used to transmit information in the opposite direction. Considerable care must be taken with the software for half-duplex lines to avoid *contention* problems, where both ends are transmitting simultaneously, and *lockup* problems, where both are expecting the

other to send the next message.

There are two main types of transmission, asynchronous and synchronous. In asynchronous communication the data is sent a character at a time, the character being composed of a fixed number of bits preceded and terminated by control bits. The control bits at the beginning of the character are called the start-bits and those at the end are called the stop-bits (Fig. 1). The time between bits within a character is constant, but the time between characters is variable. Asynchronous communication is most commonly used for communication with slower speed human-operated terminals.

In synchronous communication, the message is sent as a continuous string of fixed-length characters. The message is preceded and terminated by control characters; it may also have control characters embedded within the message [Fig 1 (b)]. The message starts with a string of synchronization characters that are used by the receiver to synchronize his clock with that of the sender. Synchronous transmission is used in those instances where a message can be transmitted in one burst, such as is normally done in intercomputer communication. The advantage of synchronous over asynchronous communication is that more useful data bits can be sent in a given time, since synchronous characters do not have any control bits.

The unit used for measuring line speed is the "baud," which is the number of information units transferred in 1 sec. The most commonly used information units are two-level (binary) units in which the value of a unit is either 1 or 0. For binary information units, the baud rate is equal to the number of bits transmitted per second. Baud rates in common use range from 110 to 50,000. This wide variation in speed provides interesting hardware/software trade-offs in the design and configuration of communication computers.

Message Transmission. The logic necessary to transmit a message will be described, using a system model in which the communications hardware in the computer is a one-bit transmit buffer that holds the communication line in either the "mark" (one) or "space" (zero) state. The approach of minimal hardware allows the message transmission logic to be described initially in software terms.

The parameters of the message transmission logic are: the directional characteristics (half- or full-duplex), the speed of the line, the transmission type (synchronous or asynchronous), and the message control and code conventions expected by the receiver. These conventions are highly application

DATA COMMUNICATIONS

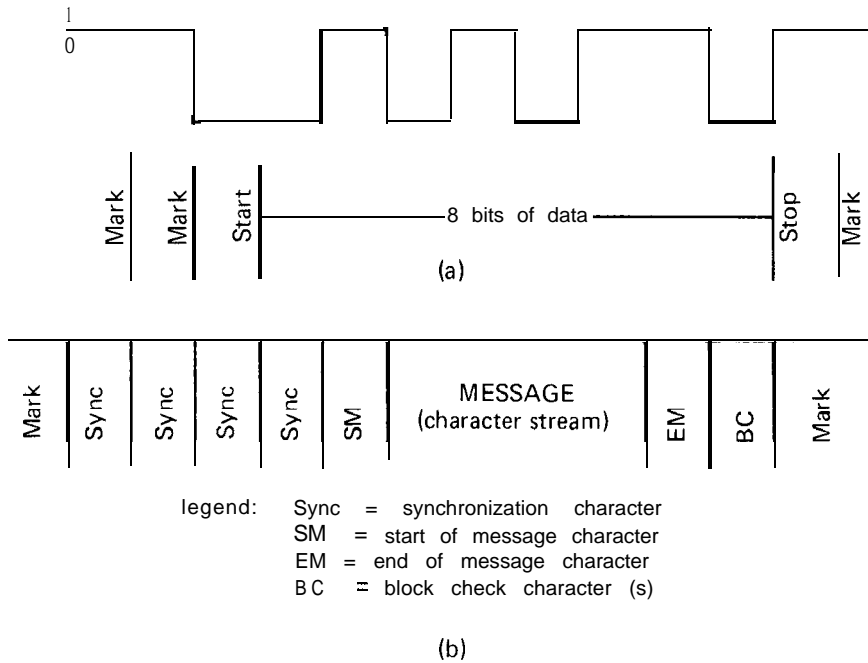


Fig. 1. Data transmission. (a) Asynchronous. (b) Synchronous.

and/or manufacturer dependent; because of this, a detailed discussion is beyond the scope of this article. We will restrict our discussion on message transmission to the generic problems associated with asynchronous and synchronous transmission.

With asynchronous transmission, each character in the message is transmitted independently. The character to be transmitted is fetched from store and framed with the required start-stop bits. The first bit (start) of this bit stream is put into the transmit buffer, which changes the line from the “mark” state to the “space” state. At the end of the one-bit time period, the next bit is put into the transmit buffer. The process is repeated until all data and stop bits are sent. If another character is to be sent immediately, then it is fetched and transmitted; otherwise, the transmit buffer is left in its “mark” state. The flowchart in Fig. 2 summarizes this logic.

In the synchronous form of transmission the entire message is sent in a one-bit stream. The message is framed by synchronization and start-of-message characters in front, and by end-of-message and error-detection characters at the rear. Then this new bit string is sent character by character, bit by bit, down the communication line. Normally the control characters are not added to the message buffer but are generated as required (Fig. 3).

Receive Function. In this section we consider the logic necessary to detect and to interpret the presence of a serial bit stream arriving on a communication line. Again we will consider minimal hardware; there is a receive buffer that gives the current state of the input line. The receive function is generally more difficult than the transmit function because the receiver has to decode and interpret the bits and characters it receives (which may be in error), whereas the transmitter knows exactly the character sequence it needs to send.

With asynchronous transmission, the character assembly logic must synchronize itself at the start of each character, since there can be a variable time between characters. To do this, the receive logic samples the line at some multiple (say, 16) of the bit rate in order to recognize the transition from the mark to space state at the beginning of the start-bit. Once the transition has been recognized, the logic waits for a half a bit time and samples the middle of the start-bit, which should still be at the space state. The logic then waits one bit time and samples the line to get the first data bit. Each succeeding bit is obtained in a similar manner and is appended to the character being formed. When the character is complete, it is processed and the logic resumes monitoring of the line for the beginning of the next

DATA COMMUNICATIONS

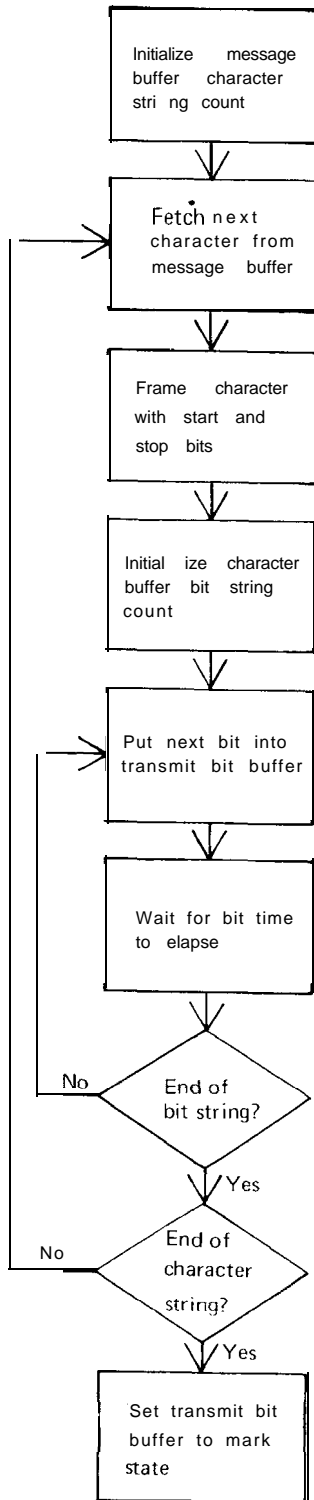


Fig. 2. Asynchronous transmit logic.

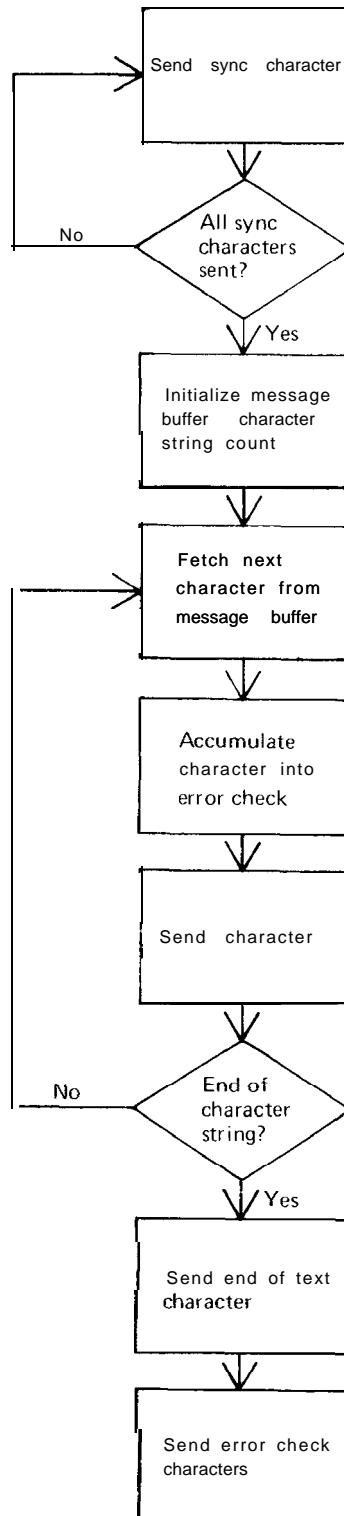


Fig. 3. Synchronous transmit logic.

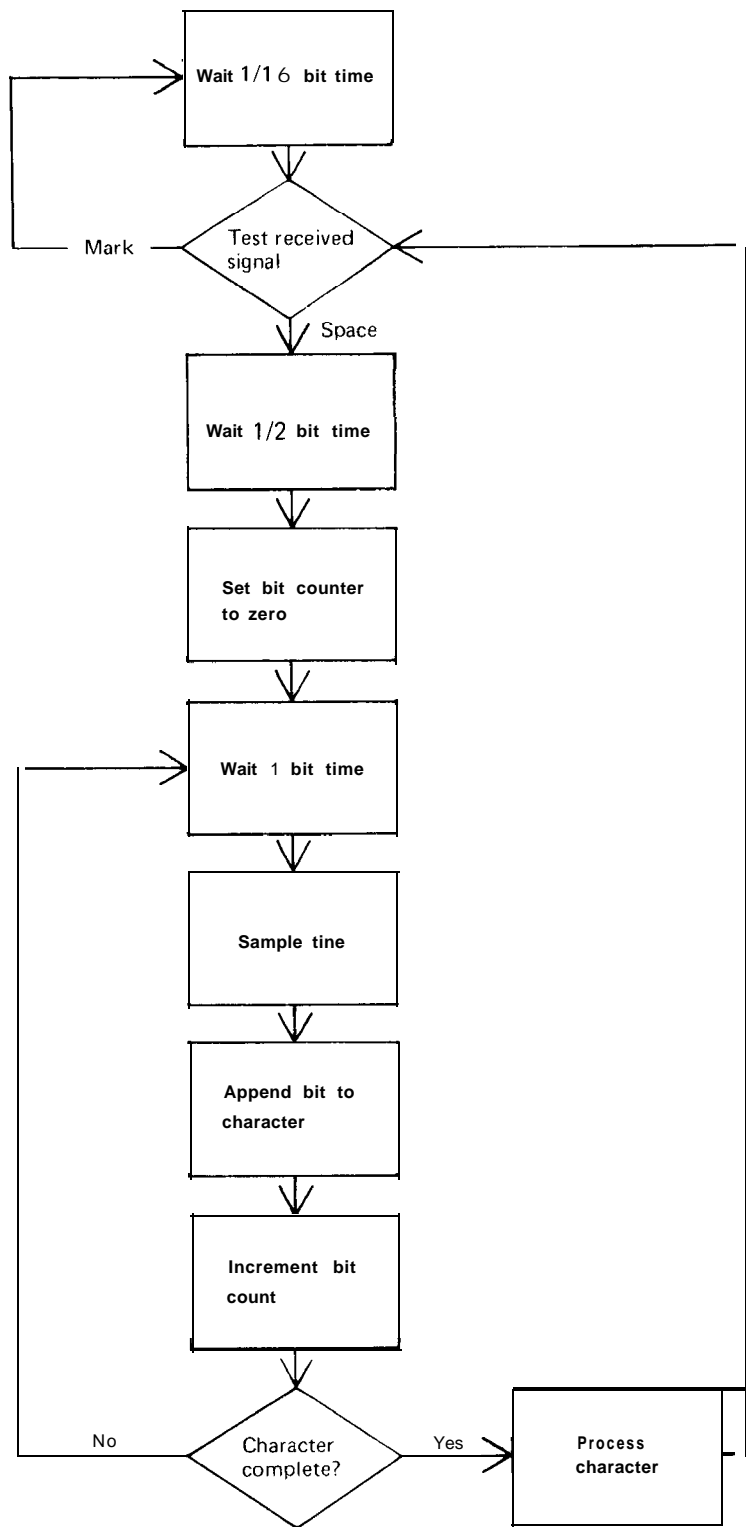


Fig. 4. Asynchronous receive function.

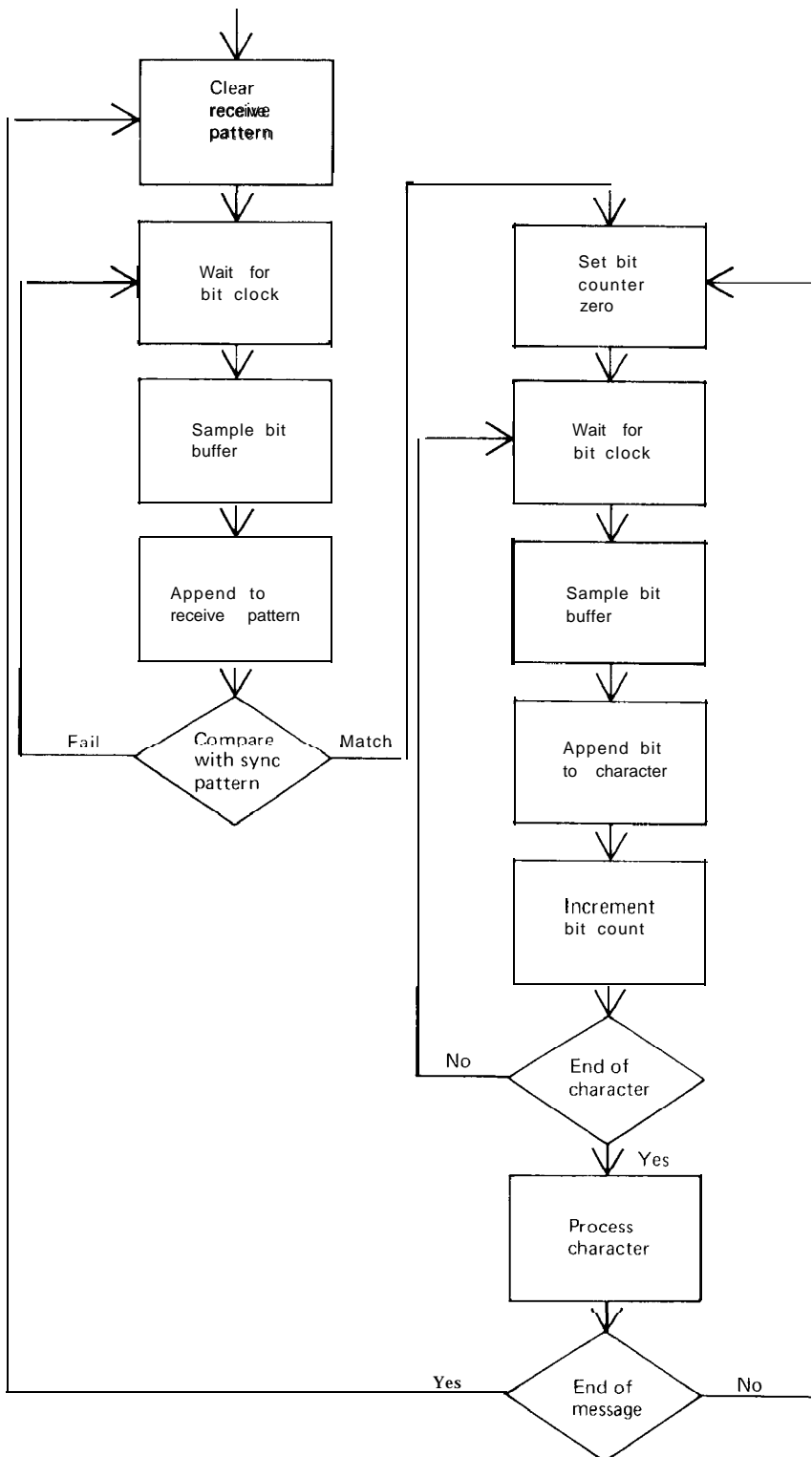


Fig. 5. Synchronous character receive.

DATA COMMUNICATIONS

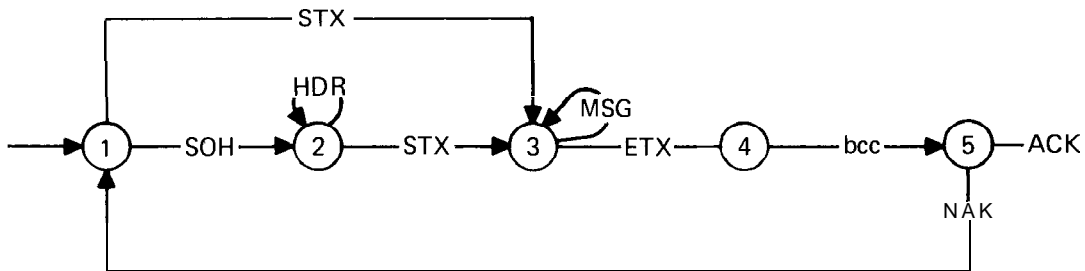
character. The flowchart in Fig. 4 summarizes this logic.

With synchronous operation, the problem is one of detecting the beginning of the message rather than the beginning of the character. The synchronous transmission scheme provides a clock signal in the middle of each bit time to control the sampling of the receive buffer. Each message begins with a fixed pattern of bits, which is used to synchronize the receiver. The start of message detection logic keeps a record of the last $n - 1$ bits received. When it gets the next bit, it compares these n bits with a fixed synchronizing pattern. If the comparison fails, then the logic waits until the next bit is received, and the process is repeated. If the comparison is good, then the next bit received will be the first bit of the first character of the message. A fixed number of bits are received and appended together to form each character. When the character is complete, it is processed. This sequence is repeated until the end-of-message character sequence is received. This action is summarized in the flowchart of Fig. 5.

After a character has been assembled, it must be processed in the context of the previous characters received. The interface between the character assembly routine and the processing routine can be a subroutine call or a queue. If the interface is a subroutine call, then the subroutine called is dependent on the line number. The direct subroutine call for character processing is the method normally used when the character processing must be completed before the next character arrival on the line. This is

known as "real time" character processing. For most applications, the character processing does not have to be done in real time. In this case, the characters together with the appropriate line number are placed in a queue for later processing (Detlefsen et al., 1972). The queue acts as a buffer and therefore allows the character processing to be spread over a longer time, thereby increasing the burst data rate that the communications computer can handle.

The processing of received characters is application dependent, but is commonly implemented using the formalism of finite state machines (Stutzman, 1972; Bjorner, 1970). A number of common line-control procedures are described in terms of the finite-state machines, which implement the discipline required for both transmitter and receiver. The input alphabet of the finite-state machine is the input character set. When a character is received (or transmitted), the machine performs some action and moves to a new state depending on the character. Typical actions performed are: put (get) character into (from) message buffer, ignore character, delete previous character in message buffer, delete message, setup to receive (transmit) error detection character(s), pass message to the routing function if no errors, setup for output after line turnaround on half-duplex lines, etc. Fig. 6 gives a finite-state machine description of a typical synchronous line discipline. For example, state 3 in Fig. 6 is entered when a start-of-text character (STX) is received, either at the start of the message or after a header has been received. Message characters (MSG) are accepted and



Legend: SOH = start-of-header character
 HDR = header information characters
 STX = start-of-text character
 MSG = message information characters
 ETX = end-of-text characters
 bcc = block check characters
 ACK = acknowledge message transmitted to sender
 NAK = negative message transmitted to sender

Fig. 6. Typical line control procedure finite-state machine diagram.

put into the message buffer until an end of text (ETX) character is encountered.

Error Detection and Correction. The ability to detect and correct errors is determined by the code set and by the transmission disciplines. The details of the error detection and correction techniques tend to be application dependent, although there are some general principles,

The simplest error detection available at the character level is **parity checking**. Parity checking is obtained by including an additional bit (the parity bit) in each character. The parity bit is a function of the data it accompanies. The transmitter generates this parity bit, and the receiver checks incoming characters for correct parity. When it encounters an error, a message may be sent to the transmitter to request retransmission.

For human-operated terminals connected over full-duplex lines, the echo-plex technique is often used. With such a terminal, the keyboard is connected only to the transmitter, and the display device is connected to the receiver. When a key is struck, a character is sent to the receiver at the other end of the line. The receiver retransmits the character back to the terminal, which displays the character typed. The operator can check to see that the character displayed is the same as the character typed.

The second level of error detection and correction applies to the message as a whole. With this scheme, the message is followed by some number of check bits, which are generated at the transmitter as a function of the characters in the message. The receiver applies the same function to the received information to generate a corresponding set of check bits that are compared with those generated by the transmitter. There are several generating functions in use today, but all can be obtained from the theory of cyclic polynomial codes (Peterson, 1961). These polynomial schemes have the capability of correcting as well as detecting errors, but error correction is normally done in practice by software-initiated retransmission. These message-oriented error control schemes are transmission independent, but they are mainly used with synchronous transmission.

Message Switching. The message switching logic used in computer-controlled networks is similar to that employed in manual systems; i.e., if the source and destination points are not physically connected by a communication path, then the message is transmitted to an intermediate point (a communication computer) and then retransmitted to

another intermediate point or its final destination.

When a message first enters the network, a message header is added to control its flow through the network. The header is removed before the message is delivered to its final network destination, normally a terminal or some EDP program external to the network.

The header contains the network destination, the message priority (which is normally application dependent), and a unique message identification. Message priority is used to determine the order in which messages are switched through the network. The identification is used for retransmission and identification. If the communications logic finds that it does not have an available path either to the final or to an intermediate destination of the message, then the message and its header must be queued within the computer until a path becomes available. In some networks, messages are switched as entities; in others, such as the ARPA network (Detlefsen et al., 1972), messages are broken into "packets" for switching purposes; and in other networks, messages are combined together for switching purposes.

Hardware/Software Trade-Offs. In the preceding sections, the transmit and receive logic has been explained in software terms, assuming minimal hardware. Flow diagrams similar to those in Fig. 2 through 5 have been used as the basis of an all software implementation on machines such as the DEC PDP-8. At the other extreme, there are hardwired communications controllers, such as the IBM 270X with hardware-implemented logic.

The main advantage of an all-software implementation is flexibility in the mix of line speeds, character sets, line disciplines, and terminal types; the disadvantage is in performance loss, particularly at higher data rates. Fig. 7 depicts hardware/software trade-offs by speed.

Hardware can replace software on several levels to improve performance. The first level adds a hardware bit buffer and a software loadable timer register that produces an interrupt when it counts down to zero. With this hardware, the character assembly/disassembly logic is done in the interrupt routine, while the character processing is done by a main program. The significant parameter in such a system is the fraction of time spent in the interrupt routine servicing the bit buffers.

The second level adds hardware shift registers (character buffers) to do the bit-to-character assembly on input and character-to-bit disassembly on output. When combined with a start-bit detector for asynchronous channels or a synchronization pattern

DATA COMMUNICATIONS

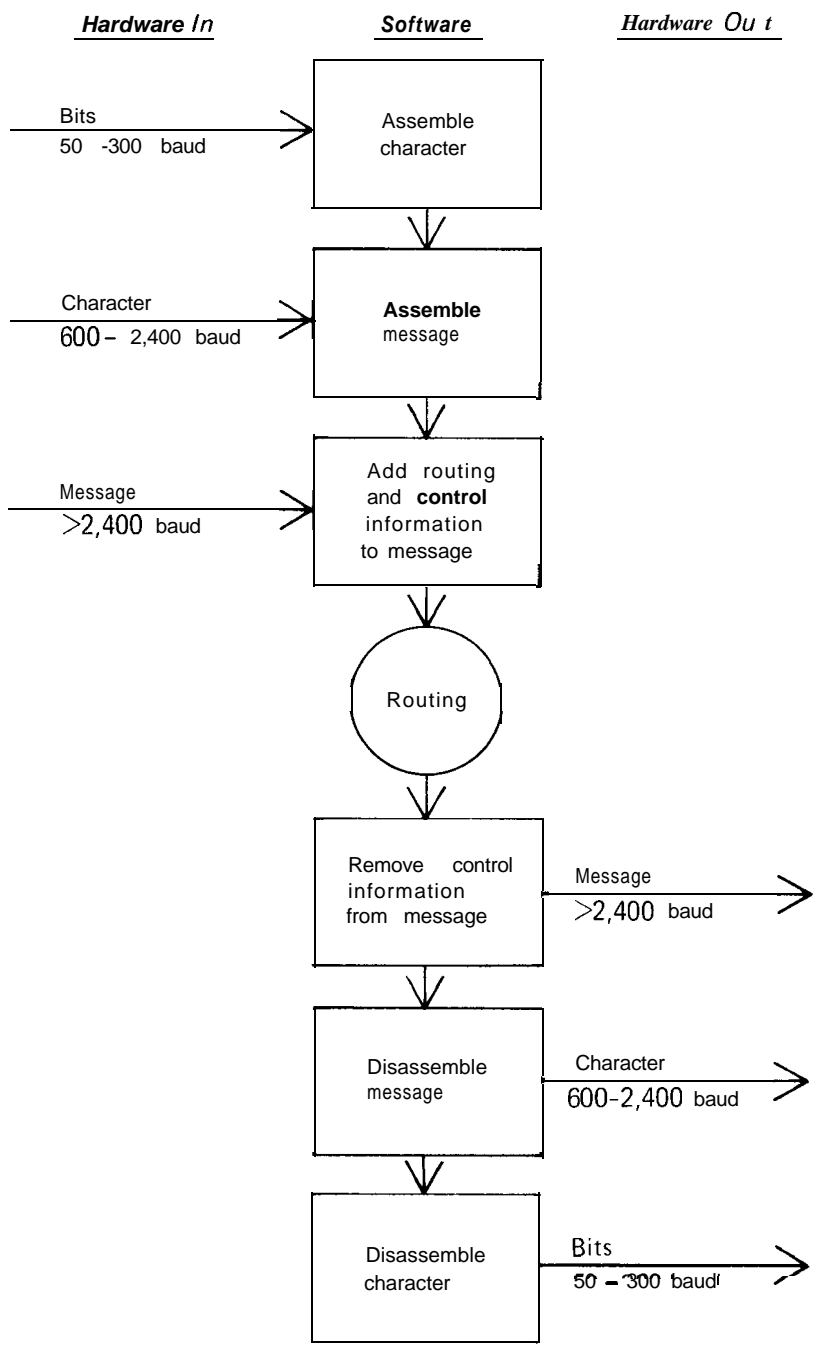


Fig. 7. Hardware/Software tradeoffs by speed.

detector for synchronous channels, these character buffers can reduce the interrupt-program processing load by a factor approximately equal to the number of bits per character. Because of their high software processing load, the polynomial error control procedures are also frequently implemented in hardware at the character buffer level. Hardware to implement circular queues in memory has been combined with character buffers in some systems (PDP-11) to reduce the processing load even further.

The third level is direct memory access (hard-wired) communication controllers. The hardware in these systems, in addition to transferring the characters to and from memory, must also perform such functions as character parity generation and checking, control character interpretation, and generation of automatic transmission synchronization sequences. The use of hardware to perform these operations reduces the software load considerably, particularly on high-speed ($>2,400$ baud) channels. Initial implementations (e.g., IBM 270X) of such channels were constrained to operate with the fixed hard-wired message and line discipline. Later implementations (e.g., Honeywell 355) have removed this constraint by allowing the message and line control parameters to be initialized under software control.

Frequently, the three major options of bit buffers, character buffers, and direct memory access controllers are available on the same communication computer and these options provide the user with the ability to configure systems that have widely varying performance and costs.

Conclusion. This article describes the major communication software functions. It does not discuss dial-up lines; multipoint lines (Stutzman, 1972); automatic recognition of line speeds; the operating system requirements for communication computers (Detlefsen et al., 1972; Sobolewski, 1972); code conversion; data compression; line monitoring, the data communication requirements on the operating system of EDP computers, or the software content of network traffic analysis and control (Cerf and Naylor, 1972; McKenzie et al., 1972).

REFERENCES

1961. Peterson, W. W. *Error Correcting Codes*. Cambridge, Mass.: The M.I.T. Press.
1970. Bjorner, D. "Finite State Automation-Definition of Data Communication Line Control Procedures," *Proc. FJCC*.
1970. Roberts, L. G., and B. D. Wexler. "Computer

- Network Development to Achieve Resource Sharing," *Proc. SJCC*.
1972. Cerf, V., and W. Naylor. "Selected ARPA Network Measurement Experiments," *Proc. 6th IEEE Computer Society Conference*.
1972. Detlefsen, G. D., R. H. Kerr, and S. B. Revkin. "Software for Data Communication Networks," *Proc. 5th Australian Computer Conference*.
1972. McKenzie, A. A., B. P. Cosell, J. M. McQuillan, and M. J. Thrope. "The Network Control Center for the ARPA Network," *Proc. First International Conference on Computer Communications*.
1972. Sobolewski, J. S. "Programmable Communication Processes," *Proc. First International Conference on Computer Communication*.
1972. Stutzman, B. W. "Data Communications Control Procedures," *ACM Computing Surveys*, Vol. 4, No. 4.

G. D. DETLEFSEN AND R. H. KERR

DATA PREPARATION DEVICES

For articles on related subjects see **AUDIO RESPONSE TERMINAL; CODES; DATA ACQUISITION COMPUTER; DATA COMMUNICATIONS: General Principles; INPUT-OUTPUT DEVICES; OPTICAL CHARACTER READERS; OPTICAL MARK READERS; and TERMINALS**.
For article on related term see **BUFFER**.

This article discusses the major data preparation devices including the card punch and paper tape punch through to the magnetic tape encoder, key-to-tape systems, key-to-disk systems, on-line key-punch/verify systems, magnetic character readers, optical character readers, optical mark readers, and direct-entry terminals. Each is briefly described together with its limitations.

The various data preparation devices and their associated data entry techniques can be separated into two main categories:

1. *Transcriptive data entry:* This term covers all data preparation devices where data, prepared on documents at their source or origin, is then transcribed to another medium that is capable of being read and interpreted by a computer. In this category are the following data preparation devices: card punches, paper tape punches, magnetic tape en-

DATA PREPARATION DEVICES

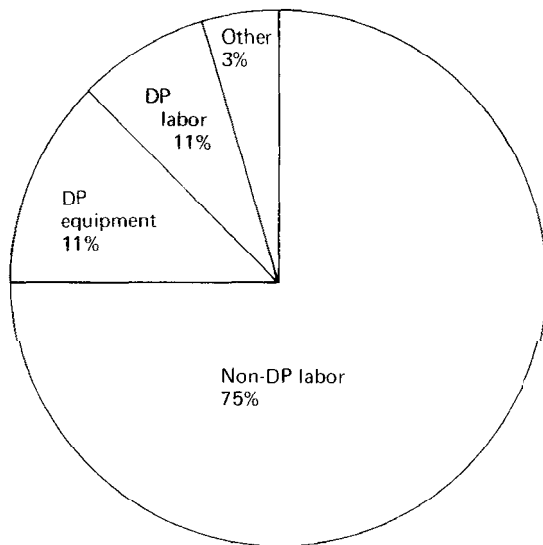


Fig. 1. Typical data-entry cost breakdown. Item distribution: DP equipment—computers, peripherals, unit record, DE equipment; DP labor—KP operations, DP operations, DE programming; other—cards, magnetic tape, forms, contracted work, card storage; non-DP labor—initial recording, coding, batching, file reference, document handling.

coders, key-to-tape systems, key-to-disk systems, on-line keypunch/verify systems and magnetic character readers.

2. **Source data entry:** For devices in this category, data is prepared at its source in a machine-readable form such that it can be directly read by a computer without the requirement for a separate intermediate data transcription step. Data entry techniques that fall into this category include optical character reading, optical mark reading, and the direct-entry of information into a computer using terminals at the point of origin of the data.

We will first examine the advantages and shortcomings of these two categories of data entry.

Transcriptive Data Entry. Each extra step carried out on data before it finally enters the computer for processing introduces the possibility of the occurrence of errors. Studies have shown that of all of the errors detected in data by the computer, only about 15% of those errors occur in source data content, with the remaining 85% introduced through

data transcription.

In order to reduce the number of errors occurring through data transcription, a number of techniques may be used. As data is transcribed into a machine-readable form, the data preparation device may carry out certain checks according to predefined conventions established for that particular data. Some of the editing that can be carried out (Trimble and Penta, 1970) includes:

1. Check digit validation.
2. Field-length check.
3. Check numeric-only fields.
4. Check alphabetic-only fields.
5. Develop batch control totals.

Data that cannot be validated by using check digits or control totals may be verified by keying it a second time, using a verifier unit to check that the second entry of the data agrees with the original entry.

The computer itself may carry out more extensive editing of the validity or reasonableness of information, applying various logical rules to the data, and possibly accessing other information held by the computer on disk or other files for confirmation purposes.

Errors that are detected must be corrected, using the original source document, rekeyed, verified again if necessary, and merged with the original data (Finklestein, 1970). The data is then edited by the computer once more, with errors recycled through the above steps until the data is error-free. Only then is the data ready for computer processing.

In addition to being involved, time consuming, and error prone, transcriptive data-entry techniques require highly trained and highly paid personnel. Indeed, salaries of such personnel can represent as much as 80% of the the total cost of data preparation (Finklestein, 1970). Moreover, these salaries are steadily increasing.

Source Data Entry. Studies of the cost of data entry show that while differences occur across companies and industries, approximately 75% of the cost of data entry occurs in nondata-processing personnel labor and delays in availability of data to the computer (see Fig. 1). Of the remaining 25% approximately 11% represents the cost of data processing and data entry equipment, approximately 11% represents the cost of data processing personnel, and approximately 3% represents other costs such as stationery and supplies.

Data-Entry Work Flow	Card and Paper Tape Punch, Magnetic Tape Encoder, Key-to-Tape System	OCR, OMR, MICR (partially)	Key-to-Disk System, On-Line Keypunch/ Verify System	Direct-Entry Terminal System
Initial recording		X	- - - - -	X
Transport		- - - - -	- - - - -	X
File reference	- - - - -		X	X
Control			X	X
Transcribe	X	X	X	X
Verify	X	X	X	X
DP edit	- - - - -	X	X	X
System input	- - - - -		X	X
Error correction			- - - - -	X

Fig. 2. Impact of data preparation devices on data-entry work flow.

The 75% cost of data entry contributed by nondata-processing personnel and time delays is expended in functions such as:

1. Retyping handwritten information for easier reading and faster operation by data preparation operators.
2. Validation, such as checking the availability of stock to fill an order before that order reaches the data preparation area.
3. Determining the price and discounts applicable for various products ordered in a preinvoicing environment .

These functions take time, with the result that before information reaches the computer, it may be several days old. Thus, the computer can produce results only as accurate and as timely as the input data. The computer in such an environment is being used only as a recording and high-speed printing machine. Its full potential cannot be realized until it is able to accept information as close as possible to the time of origination of that information.

Source data entry removes the need to retype handwritten information for easier reading by data preparation operators. In fact, the need for such data preparation is completely bypassed by entry of the data directly from its source. In addition, the computer itself can check the validity of the information.

In this way, instead of reflecting the status of information that may be several days old, the computer will maintain much more current information. Information, therefore, is available sooner, is more accurate, allows more meaningful decisions to be made, improves customer service, and reduces the amount of time before the organization will be paid for service performed.

The effect of each of the data preparation devices discussed above on the data-entry work flow is summarized in Fig. 2. Each step in the data-entry cycle affected by a particular device is illustrated in Fig. 2 by a cross, with those devices having a similar effect on the work flow grouped together. This figure illustrates the fact that only source data entry, using direct-entry terminals, has an effect on every step in the entry of data into a computer.

We will now examine each of the various data preparation devices in more detail.

Transcriptive Data Entry Devices

CARD PUNCH AND VERIFIER. Until the 1960s, the capabilities of most card punches were limited. When the punch operator hit a key on the keyboard, the appropriate combination of holes was immediately punched into the card. In many cases, errors made by keypunch operators were detected by them immediately after making the error. However, correction of such an error invariably required that the card be ejected, inserted into the read mechanism, duplicated up to the point of the error, the error corrected, and punching continued. This procedure was time consuming. Examples of these units are the IBM 24, 26, and 29 card punches, and the IBM 56 and 59 card verifiers (see Fig. 3a).

The 1960s saw the development of a buffer on the card punch so that an entire card of information could be keyed, and any error could be corrected by backspacing to the point of the error and rekeying. Only after the punch operator was satisfied with the information keyed was that information released from the buffer for punching.

Other developments provided punches with capabilities of validating check digits and accumulat-

DATA PREPARATION DEVICES



Fig. 3. Data preparation devices. (a) IBM card punch and verifier. (b) Data Action 150 magnetic data inscriber and IBM 2495 tape cartridge reader used to transfer data entered on the Data Action 150 into a computer.

ing information for comparison against batch control totals (Trimble and Penta, 1970). The 1960s also saw the emergence of combination devices for both card punching and verifying, thus enabling a key-punch operator to correct a punching error on the same device used for verification. Examples of such units are the Univac 1701 verifying punch, the Univac 1710 verifying interpreting punch, and the IBM 129 card punch.

PAPER TAPE PUNCHES. Paper tape punches permit data to be entered without the restriction of 80 or 96 columns for information on a card. Thus, records of information relating to a particular transaction can contain as much or as little information as necessary, without the physical limitation imposed by the card.

When errors are introduced by the punch operator, the error information on the paper tape is backspaced over, erased by punching a series of delete characters (which are ignored when read by a paper tape reader), and then rekeyed. In many cases, verification of the correction is not used with paper tape; instead, that data is read directly by a computer and edited. In the event of errors being detected, corrections are keyed generally by punching a reversing transaction for the information in error, and then punching the correct information. In addition, information that may have been omitted can be punched on a separate piece of paper tape and then spliced in sequence into the main section of the tape.

Most computer manufacturers like NCR, ICL, Burroughs, CDC, and Univac produce paper tape punches.

MAGNETIC TAPE ENCODERS. The first magnetic tape encoder was announced by Mohawk Data Sciences in 1965, and gained immediate acceptance.

While the magnetic tape encoder can be used for the transcription of data directly to magnetic tape (Finkelstein, 1970; *EDP Analyzer*, Part I, 1971), most magnetic tape encoders such as the Mohawk encoder also have the ability to transmit data from one point to another over telephone lines. Thus, they are well suited for the preparation of data in remote locations, and the transmission of that data to a central point where it may be received by another encoder, recorded directly on magnetic tape, and then used as input into a computer.

As well as recording data directly on half-inch computer magnetic tape, as with the Mohawk encoder, other units have been developed to record data on tape cassettes or cartridges. These cartridges are converted to half-inch computer tape, or read directly into a computer, by a cartridge reader,

Examples of these units are the IBM 50 magnetic data inscriber and IBM 2495 tape cartridge reader, the Viatron System 21, and units manufactured by Sycor and by Data Action (Fig. 4).

While magnetic tape encoders remove the need to use cards or paper tape, and feature a buffer for easy correction of keying errors, they still suffer from most of the disadvantages of card and paper tape punches. The data preparation cycle is still involved and time consuming, and errors must be recycled for correction. While most encoders offer features such as check-digit validation and control total accumulation, data cannot be fully validated until it is processed by the computer edit program.

KEY-TO-TAPE SYSTEMS. In 1969, Mohawk announced the 900 Series key-to-tape system, which groups up to 16 key stations around a control unit and pools data from these stations onto magnetic tape. Honeywell produces a similar unit, the Key-tape [see Fig. 4a], as does Singer-Friden, who manufactures the 4300 Magnetic Data Recording System.

Key-to-tape systems were developed to overcome the limitations of magnetic tape encoders (*EDP Analyzer*, Part I, 1971). They comprise a number of keyboards, possibly also with television-like cathode-ray tube (CRT) displays, and are connected to a central controlling unit, typically a minicomputer, which collects information from each keyboard. This information is then directed to a magnetic tape.

In addition, the use of a minicomputer allows more sophisticated validation and editing to be carried out at the time of initial entry of the data. Consequently, more errors can be detected earlier than is possible with the devices discussed above, and correction of these errors is simplified.

However, such key-to-tape systems generally do not have the capabilities of accessing other computer files for more complete validation of data. Consequently, errors must still be recycled from the computer edit run for correction. Data preparation is still an involved, time-consuming process.

KEY-TO-DISK SYSTEMS. Key-to-disk systems are effectively equivalent to key-to-tape systems, except that information entered by keyboards is first collected on magnetic disk (*EDP Analyzer*, Part II, 1971). Each keyboard generally uses a separate section of the disk, and is independent of other keyboards. Thus, each keyboard can be working on different types of data at the same time, which results in good operational flexibility in the installation. Data is extracted from the disk when complete, and is copied onto a magnetic tape or another

(a)



(b)

Fig. 4. Data preparation devices. (a) Honeywell KEYPLEX system allows up to 64 operators at KEYTAPE stations to encode data onto magnetic tape. (b) Computer Machinery Corporation (CMC) 18 key-to-disk system can support up to 64 input stations.

disk for further processing on the main computer,

Some key-to-disk systems, such as the IBM 3740 data entry system, are also capable of transmitting data from the disk to a device that copies the data to magnetic tape for later processing, or transmits it across telephone lines directly to a computer. Examples of key-to-disk systems are the CMC18 Key Processing System, manufactured by Computer Machinery Corporation of Los Angeles [see Fig. 4(b)], the Inforex Key Entry System manufactured by Inforex Inc. of Burlington, Massachusetts, the Honeywell Keyplex System, the Logic Corporation Key Disk System, the IBM 3735 programmable buffered terminal, and the IBM 3740 data entry system.

Most key-to-disk systems generally have limited disk capacity so that they cannot also hold full computer files to allow complete validation and editing. This validation against computer files still must be left until the main computer edit run. Accordingly, errors must still be recycled for correction.

ON-LINE KEYPUNCH/VERIFY SYSTEMS. On-line keypunch/verify systems, such as IBM's VIDEO/370 program product, provide a software capability similar to that of key-to-disk systems (Finklestein, 1970), but are controlled by a main computer rather than a minicomputer. Consequently, they offer the potential for validation against full computer files, so enabling complete editing of data without requiring a separate computer edit run. In this way, error correction is considerably simplified.

MAGNETIC CHARACTER READERS. Magnetic ink character recognition (MICR) is a transcriptive data entry technique used mainly by banks. Information, such as the amount of a check, is transcribed and printed in magnetic ink when the check is accepted by the particular bank. Examples of magnetic character readers are the IBM 1419 (see Fig. 5), IBM 1259 and IBM 1255 (see Fig. 6), the Burroughs OCR/MICR reader/sorter, and the Honeywell 232 reader/sorter.

Magnetic character readers are designed to be used off-line, away from a computer, for certain editing of information and physical sorting of checks, and also on-line to a computer for direct entry of information from checks. After the computer edit run, errors must still be recycled for correction.

TRANSCRIPTIVE DATA ENTRY SUMMARY. It can be seen from the above discussion that developments in transcriptive data-entry devices have been directed toward increasing the amount of editing that can be carried out when data is initially transcribed

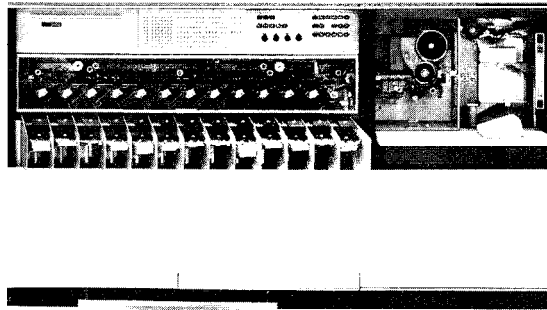


Fig. 5. IBM 1419 magnetic character reader.

to a machine-readable form. These developments have also reduced the delays that occur in validating information and correcting errors, and have resulted in up to 30 to 50% increased throughput of key-to-disk systems and on-line keypunch/verify systems over card punching (*EDP Analyzer*, Part II, 1971). However, as was discussed earlier, transcriptive data-entry techniques apply only toward the 25% of the total data-entry cost that is represented by data-entry equipment, data processing personnel labor costs, and supplies (see Fig. 1). Consequently, the net effect of newer devices on the total data-entry cost is an effective increased throughput on the order of 8 to 12%.

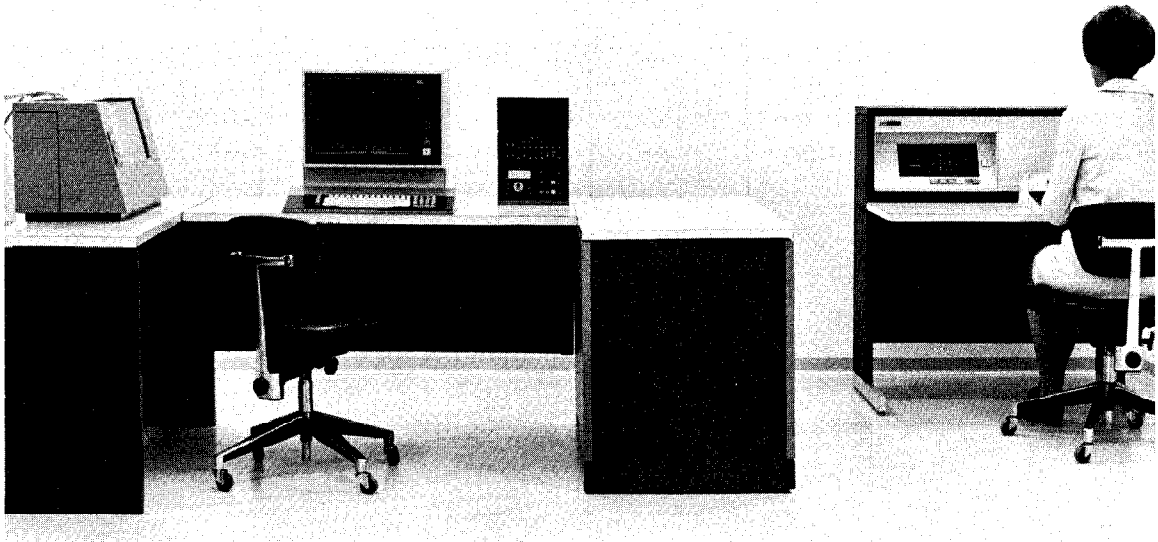
Source Data Entry

OPTICAL CHARACTER READERS (OCR). Since typed or numeric hand-printed information capable of being read by OCR's is also readily understood by humans, this is a very useful source data-entry technique. The stylized alphabetic fonts recognized by an optical character reader can be printed by normal electric typewriters at the point of origin, and the need for transcriptive data entry is eliminated (Finklestein, 1970.) This also eliminates the possibility of introducing additional errors, as well as eliminating the high cost of data transcription equipment and personnel.

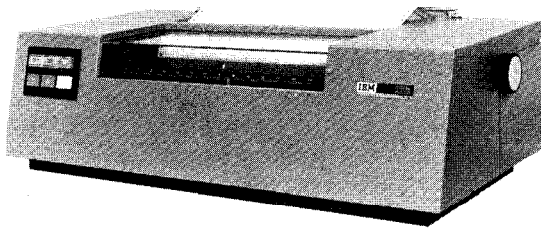
A large number of OCR's are now available. These include units manufactured by Farrington (3030), Recognition Equipment Inc., Univac, Control Data Corporation (915 and 955 page readers), IBM (1287 and 1288 optical character readers), Optical Scanning Corporation (288 document reader), Honeywell, Scan-Data Corporation, Scan-Optics Inc., and Viatron.

Information printed by the computer can be also used as a turnaround document, with additional

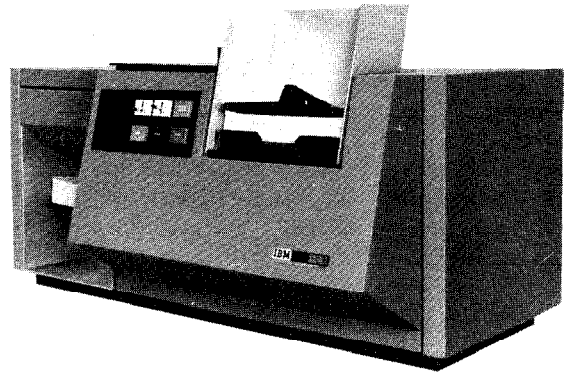
DATA PREPARATION DEVICES



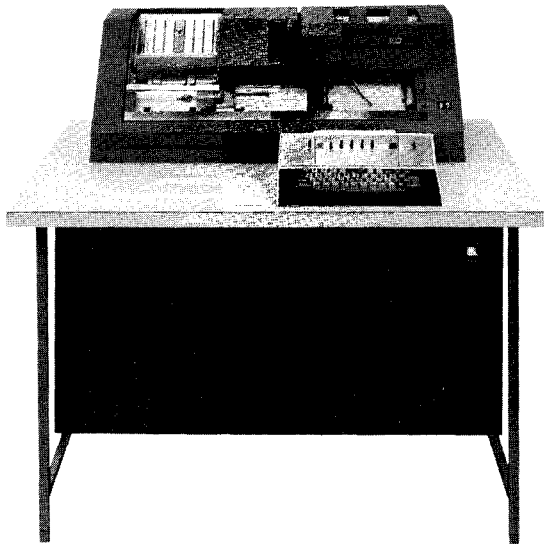
(a)



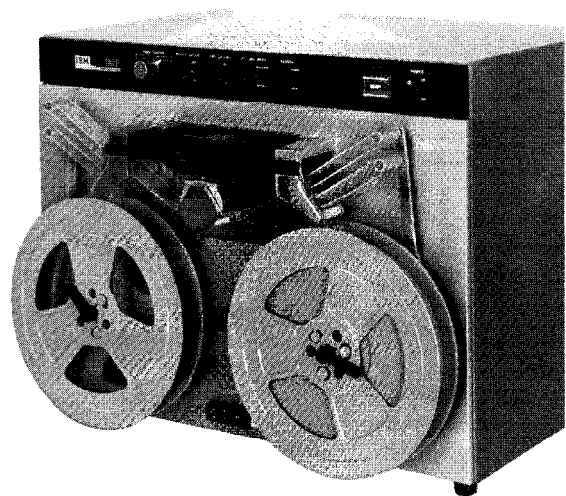
(b)



(c)

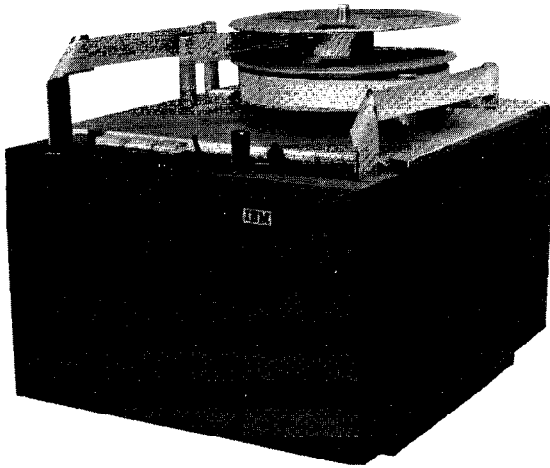


(d)

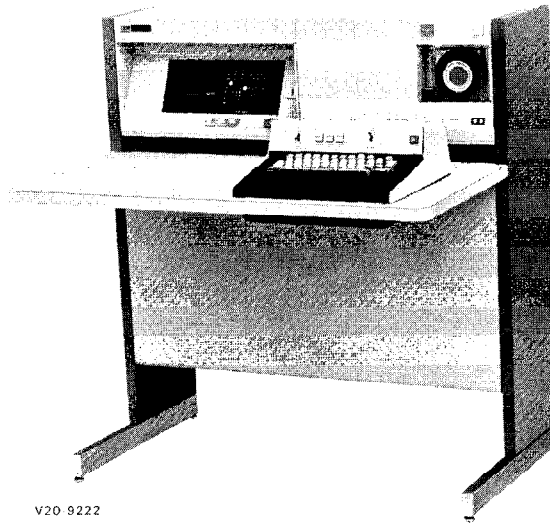


(e)

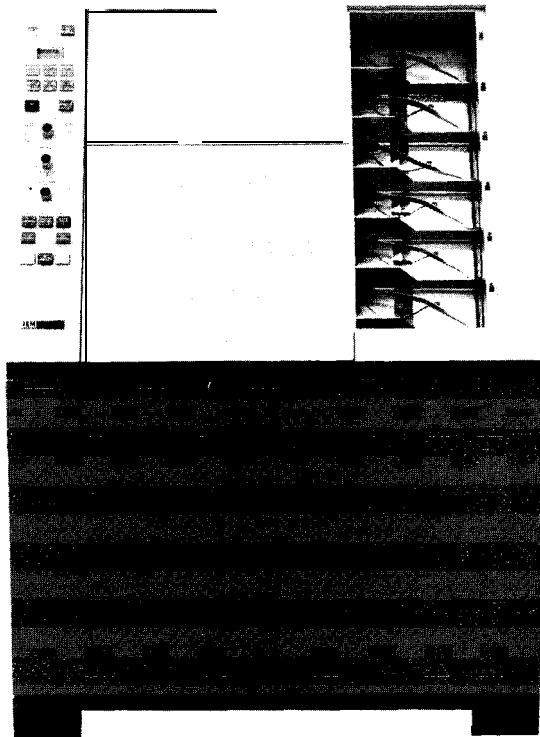
DATA PREPARATION DEVICES



(f)



(g)



(h)

Fig. 6. Components of the IBM 2770 data communications system. (a) 2772 Multi-Purpose Control Unit and Keyboard, 2265-2 Display Station, 2213-2 Printer; (b) 2213-1 Printer; (c) 2502 Card Reader; (d) 545 Output Punch; (e) 1017 Paper Tape Reader; (f) 1018 Paper Tape Punch; (g) 50 Magnetic Data Inscriber; (h) 1255 Magnetic Character Reader.

DATA PREPARATION DEVICES

numeric information being hand printed, using a pencil. Examples of such turnaround documents are insurance policy renewal forms, meter-reading forms, credit authorizations, and so on.

Optical character readers are generally connected directly to a computer, and read and edit documents at high speed. Because of the input speed, computer files generally cannot be accessed for full editing. Any error documents are selected into a separate stacker and must be recycled for correction.

OPTICAL MARK READERS. Optical mark readers (OMR) enable data to be recorded using a soft pencil as a series of marks on a sheet of paper (Finklestein, 1970). Each mark represents information according to its position on the page.

Optical mark readers may be connected off-line to a card punch or magnetic tape, or on-line directly to a computer. Because of the relatively slow speed of such readers, information may be fully edited during data entry, including access to computer files for complete validation. Error sheets can be selected into a separate stacker, corrected by erasure of the error mark, and then recycled.

Examples of optical mark readers include the IBM 1230, 1231, 1232 and 3881, the Optical Scanning Corporation 70 and 100, the Scanak 216, and the Republic Corporation 1500 optical card scanner.

Optical mark readers enable data to be captured very economically directly at its point of origin in a machine-readable form. The only equipment needed is a soft lead pencil and a supply of preprinted optical mark forms. The training necessary to record information accurately for optical mark reading can be carried out in a matter of minutes.

This technique is ideally suited for applications such as surveys, questionnaires, and diagrammatic representation of information, as well as numeric and alphabetic information. Turnaround documents, produced by the computer, can also be used for subsequent input, using optical mark readers.

DIRECT-ENTRY TERMINALS. These terminals enable the computer not only to receive data immediately after it has been entered at its point of origin, but also to edit the data at its time of receipt, allowing computer files to be accessed to validate the information entered, and allowing the terminal operator to be notified immediately of any errors.

Some information (such as dollar values or sales figures) may not be able to be checked against computer files. Such information may instead be verified by rekeying, as for card punches, and the computer can check that the same information was entered each time. While verification may still be

necessary in these cases, the amount of data to be verified is generally reduced. Thus, the correction process is greatly simplified, and the time delays associated with the transportation of data from the source as well as the detection and correction of errors are almost eliminated.

A variety of devices can be attached to terminals for communication with a computer. Some of these devices are shown in Fig. 6, which illustrates components that may be attached to the IBM 2770 data communications system.

The power of the computer can also be used to provide information that might not otherwise normally be available at the time. An example of the advantages of source data entry is given by examining the direct entry of orders from terminals in a branch office. An order is entered directly into a terminal at the time it is placed by a customer. Computer customer and product files can be accessed, and the quantity of each product ordered can be checked against the available quantity-on-hand in the product file, and updated. If insufficient stock is available to fill the order, the computer may automatically place the remaining quantity to be supplied on backorder. The price and any relevant discounts of each product ordered can be used to determine the total value of the order, which can then be checked against the customer's credit limit.

When the order has been completely entered, the computer may transmit it directly to a terminal in the warehouse. The products ordered may be sorted by the computer into the same sequence as they are located in the warehouse, to enable a packing slip to be produced on the warehouse terminal. At the same time, a completely calculated invoice may also be produced for inclusion with the order as it is packed, so enabling pre-invoicing to be readily implemented. In addition, a copy of this invoice can be transmitted back to the branch office terminal as confirmation that the order has been accepted. In the same way as stock is received from various suppliers to be placed in the warehouse, these receipts can be entered from the warehouse terminal to update the quantity-on-hand file for each particular product received.

It can be seen from the preceding example that the potential offered by terminals used for source data entry is enormous. The transcription requirement, with its consequent delays and cost, is eliminated, and information is available to the computer at its time of origin rather than several hours or days later. Thus, the computer is able to reflect the actual operation of the company accurately and reliably.

However, the additional cost of data transmission, direct-entry terminal, and computer equipment is a factor that must also be considered.

We will now consider various terminal devices used for source data entry.

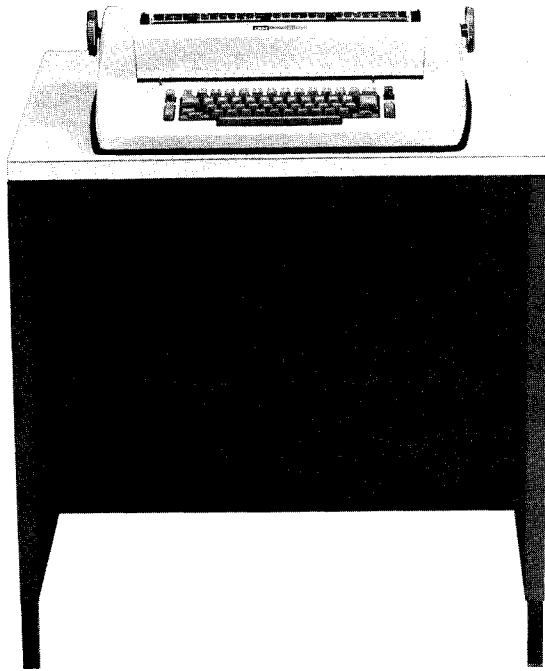


Fig. 7. IBM 2741 typewriter communications terminal.

Typewriter Terminals. These devices range from the normal office Telex machine used for telegraph communication to terminals that can be used as normal office typewriters when not required as terminals, such as the IBM 2741 (see Fig. 7) and the IBM 2740 communications terminals. Using typewriter terminals, a hard-copy record of information entered via the keyboard and received from the computer via the printer can be recorded on paper for subsequent reference.

In order to share the cost of communication between a number of terminals, typewriter terminals often may all attach to (or be "multidropped" off) the same communication line. In this instance, in order to avoid a tie-up of the line for relatively long periods of time while a terminal operator keys information, such terminals often feature buffers into which the data can be keyed. Once that data has been keyed, it can be input to the computer for

transmission at line speed when the line is idle and available for use. The IBM 2740-2 communications terminal is an example of such a buffered terminal.

Visual Display Terminals. Visual display terminals, or cathode-ray tube (CRT) terminals as they are sometimes called, generally use a keyboard for entry of information, a television-like screen for the display of that information, and an optional printer for hard copy (see Fig. 8). They are generally characterized by a higher speed of operation than typewriter terminals. Because of the widespread acceptance of such devices, visual display terminals are produced by almost every computer and terminal manufacturer, such as IBM (2260, 2265, and 3270), Univac, NCR, CDC, and Burroughs.

Visual display units (Fig. 9) are well suited as both inquiry and data preparation devices (Finkelstein, 1970). In the event of an error being detected in data entered from the terminal, the computer may display an error message on the screen, and also lock the keyboard and sound an audible alarm to notify the terminal operator (Finkelstein, 1970).

As with typewriter terminals, most visual display terminals may be multidropped off the same communication line. Visual display terminals use a buffer to enable information to be continually regenerated to maintain the image on the screen. This buffer is also useful to enable data to be transmitted at communication-line speeds, thereby sharing that line with other visual display terminals.

Audio-Response Terminals. Audio-response terminals feature either a full typewriter keyboard (such as the IBM 2721 portable audio terminal) or only a numeric keyboard, and do not use a printer for output. Instead, the response transmitted from the computer is a spoken response. The computer program can edit the data entered from the keyboard, process that data and construct a reply. This response is directed to an audio-response unit attached to the computer, which converts it to a spoken reply. This spoken response is then transmitted over the telephone line to the audio-terminal speaker. The Touchtone telephone in household use today may be used as an audio data-preparation device. These telephones enable numeric information to be keyed, once the call has been established.

Audio-response terminals may be used for source-data-entry applications (Finkelstein, 1970), such as the order-entry example discussed earlier. In this case, each product number and quantity ordered can be keyed on the audio terminal and the confirmation of acceptance of the order, or any errors can be transmitted back by the computer as a spoken response. The advantage that such audio-

DATA PREPARATION DEVICES



Fig. 8. IBM 3270 information display system, showing display screens and printer.

response terminals have over typewriter terminals is their ready availability. Thus, a representative from a company may travel to different customers in his territory, accept orders from the customer on his premises, telephone the computer, and enter those orders directly, using the customer's own telephone.

Intelligent Terminals. An intelligent terminal generally has the ability to store data on magnetic tape or disk, and is also able to carry out extensive formatting and editing of information as data is entered and before it is transmitted to the computer. Thus, various edit checks may be carried out as data is entered, and errors may be corrected at that time (Fig. 9). However, it is not possible to carry out full editing by validating information against files until the data is transmitted to the computer at a later time. Therefore, intelligent terminals require the recycling of error corrections, as for transcriptive data preparation devices. However, they offer an

effective compromise between the full advantages to be gained through source data entry, and the cost of transmission of large volumes of data across long distances.

Some examples of intelligent terminals are the IBM 3735 programmable buffered terminal, the Burroughs TC500, and minicomputers such as the PDP 8 Series manufactured by Digital Equipment Corporation.

REFERENCES

- 1970. Trimble, Jr., G. R., and A. J. Penta. "Evaluation of Keyed Data Entry Systems," *Data-mation* (June), pp. 93-99.
- 1970. Finkelstein, C. B. "Data Entry Techniques," *Australian Computer Journal* (November), pp. 146-155.

DATA PROCESSING

For articles on related terms see **CENTRAL PROCESSING UNIT**; **DATA COMMUNICATIONS**; General Principles; **GENERATIONS**, **COMPUTER**; **STATISTICAL APPLICATIONS**; and **TIME SHARING**.

Data processing is a widely used term with a variety of meanings and interpretations ranging from one that makes it almost coextensive with all of computing (e.g., IBM's major marketing division is called the "Data Processing" Division) to much narrower connotations in the general area of computer applications to business and administrative problems,

In a broad sense, data processing may be said to be what computers *do*. In this context it should be compared to *information processing*, which some prefer to data processing because "information" does not carry the connotation of "number," as "data" sometimes does. Of course the "data" in data processing is really intended to connote any kind of information in symbolic form. Thus, information may be viewed as "knowledge," while data are the physical symbols used to represent the information.

The term "data processing" is often used with various modifiers, the most common being:

1. Electronic data processing (EDP), a term widely used to describe *all* computing activity-or, at least, the part of computing that focuses on administrative or business applications-and particularly to distinguish computerized applications from manual methods.

2. Automatic data processing (ADP), closely analogous to EDP, since it is intended to distinguish computer data processing from data processing where significant human assistance or intervention is required.

3. Business data processing (BDP) refers specifically to administrative applications (e.g., personnel, payroll, accounting) and to broader business applications (e.g., inventory control, sales forecasting).

4. Scientific data processing, which is still a rather rarely used term and which is meant to imply the increasing recognition that business and scientific applications of computers have much more in common than was once realized or, indeed, than was actually the case in earlier days.

Until the 1960s it was common to divide the world of computer applications into two realms -business data processing and scientific computing -with the latter encompassing all engineering, sci-



Fig. 9. IBM 3270 information display system, being used for payroll inquiries and data entry.

197 1. "Improvements in Data Entry," *EDP Analyzer*, Part I, Vol. 9, No. 9 (September); Part II, Vol. 9, No. 10 (October).

C. B. FINKELSTEIN

DATA PROCESSING

For articles on related subjects see **ADMINISTRATIVE-BUSINESS APPLICATIONS**; **INFORMATION AND DATA**; **INFORMATION PROCESSING**; **PROCEDURE-ORIENTED LANGUAGES**; and **SCIENTIFIC APPLICATIONS**.

DATA PROCESSING

entific, or other technical applications of computers where the emphasis was on numerical calculations, usually extensive ones, rather than on the manipulation (sorting, organizing, etc.) of data (together with, at most, very simple arithmetic calculations), which was the province of business data processing.

Another distinct, although related contrast between the two areas was their relative dependence on the central processing unit facilities of the computer on the one hand and on the input-output facilities on the other hand. Most scientific calculations seemed to require little input data, produced relatively few numbers as results, but relied heavily on the arithmetic and logical capabilities of the CPU. Indeed, computers that handled mainly large scientific calculations were, and still are, often called "number crunchers." By contrast, business data processing tasks usually involved large amounts of input data (e.g., the entire employee file of a company)-hence the name "data" processing-performed relatively few calculations, and then produced large amounts of output (e.g., all payroll checks for the company).

To a degree, this dichotomy between scientific calculations and business data processing was always misleading. If the paradigm for business data processing-much input and output, little calculation-was, in fact, a rather good generalization, the paradigm for scientific calculation was much less so. Scientific calculations involving large volumes of input data and, more commonly, large quantities of results had been common since the earliest days of computing (e.g., the production of tables of mathematical functions such as the trigonometric or Bessel functions). Still, it has only been in recent years that the dichotomy has been seen to be less and less useful for any purpose.

Increasingly, scientific calculations (e.g., meteorological and high-energy physics applications) process large amounts of input data and produce copious results. Also increasingly, although less so, business applications involve sophisticated mathematical techniques involving large amounts of calculation (e.g., various statistical and related forecasting applications). Thus, while there remain many computer applications that conform to the original business data processing/scientific computing stereotype, it is increasingly common and, this author believes, more reasonable to use the terms "business data processing" and "scientific data processing" to distinguish between applications areas but not between the characteristics of the applications themselves.

The past distinction between business data processing and scientific calculations was reflected

in the development of computers ostensibly designed for one application area but not the other. IBM's 700 series of computers of the 1950s illustrates this point. (The 700 series comprised first-generation computers, which utilized vacuum tube technology; with the advent of transistor technology and the second generation of computers, a zero was added, and this became the 7000 series. Thus, the 7040 and 7090 were transistorized and somewhat modified versions of the 704 and 709.) There were two pairs of computers in this series, first the 701 and 702, and later the 704 and 705. (There was also a 709, more powerful but quite similar to the 704.)

Both the 701 and 704 were designed for scientific computing. Their memories were binary and word-oriented and, on the 704, floating-point arithmetic was standard. By contrast, the 702 and 705 were specifically designed for "data processing" applications, meaning business data processing. Their memories were character- and digit-oriented and only fixed-point arithmetic was possible. By the time of the advent of the IBM 360 series of computers in the mid- 1960s, the previous sharp distinction between scientific computing and business data processing was becoming blurred so that the existence of separate computers for the two areas was no longer considered necessary. Nevertheless the distinction still was considered important and, for example, one model of the 360 series, the 360-44, was specially designed for scientific computation.

In the 1970s some manufacturers still orient their general-purpose computer line toward particular application areas, most notably Control Data with its 6000, 7000, Cyber 70, and Cyber 170 series of computers intended mainly for scientific applications, but the trend is clearly toward computers for data processing without a distinction between scientific and business applications.

The development of general-purpose higher-level programming languages also parallels the history outlined in the preceding paragraph. The first such language in the mid- 1950s, Fortran, was intended (and still is mainly used) for scientific calculations. Even the current version, Fortran IV, lacks the significant character manipulation and good data structure facilities needed for many data processing problems. And its input/output facilities are relatively rudimentary. The second such language in the late 1950s, Cobol, was intended (and still is virtually always used) for business data processing problems. Its arithmetic facilities, lacking as they do a floating-point arithmetic capability, virtually preclude its use for significant numerical calculations.

The development of PL/I in the mid-1960s had, among its motivations, the desire to develop a language that could be used for both scientific and business problems because of increasing cognizance about this time of common properties in these two applications areas. PL/I's failure, up to the mid-1970s at least, to achieve wide popularity cannot be ascribed to any deficiency in this viewpoint. Rather, it is due to the very large inertia among Fortran and Cobol users which prevents them from switching to a new language because of their extensive investment in programs, libraries, and expertise in the older languages.

In the future we may expect the distinctions between the scientific and business applications areas to be further blurred as time sharing, widespread use of data communications, and increasing use of large data bases further pervade all applications areas. The name "data processing," therefore, will remain an inclusive term to describe computer applications of all kinds. It will continue to be one of a few terms (information processing and symbol manipulation are others) that may reasonably be used to denote what a computer does.

REFERENCE

1973. Davis, Gordon B. *Computer Data Processing* (2d ed.). New York: McGraw-Hill.

One of the better books that focuses on business applications.

A. RALSTON

DATA PROCESSING MANAGEMENT ASSOCIATION (DPMA)

For articles on related subjects see **AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES**; and **INSTITUTE FOR CERTIFICATION OF COMPUTER PROFESSIONALS**.

Purpose. The Data Processing Management Association is one of the largest worldwide organizations serving the information processing and computer management community. It comprises all levels of management personnel and, through its educational and publication activities, seeks to encourage high standards of performance in the field of data processing and to promote a professional

attitude among its members. Its specific purposes, as stated in its international bylaws, are as follows:

- A. To foster, promote and develop education and scientific inquiry in the field of data processing and data processing management.
- B. To inculcate among its members a better understanding of the nature and functions of data processing, and to engage in education and research in the technical methods pertaining thereto with a view to their improvement.
- C. To collect through research and to disseminate generally, by all appropriate means, all fundamentally sound data processing principles and methods.
- D. To study and develop improvements in equipment related to data processing.
- E. To supply to its members current information in the field of data processing management, and to cooperate with them and with educational institutions in the advancement of the science of data processing.
- F. To encourage and promote a professional attitude among its members in their approach to an understanding and application of the principles underlying the science of data processing and in their relations to others similarly engaged.
- G. To foster among executives, the public generally, and the members of the Association a better understanding of the vital business role of data processing, and the proper relationship of data processing to management.

How Established. Founded in Chicago as the National Machine Accountants Association, DPMA was chartered in Illinois on Dec. 26, 1951. At this time the first electronic digital computer had yet to come into commercial use, and the name "machine accountant" was chosen to identify those associated with the operation and supervision of punched card accounting machines. Twenty-seven chapters were organized during the Association's first year. By 1955, the organization had taken on an international character with the admission of Montreal as the first Canadian chapter.

With the rapid advances in information processing techniques brought about by the introduction of computers, the nature of the Association further changed as membership swelled from ranks of computer management. In step with this trend,

DATA PROCESSING MANAGEMENT ASSOCIATION

the Association assumed its present name in 1962. The roster of past presidents includes the following:

Robert L. Jenal, 1952	John K. Swearingen, 1964
Gordon C. Couch, 1953	Daniel A. Will, 1965
Richard L. Irwin, 1954	Billy R. Field, 1966
Robert O. Cross, 1955	Theodore Rich, 1967
Donald L. Gerighty, 1956	Charles L. Davis, 1968
Willis L. Daniel, 1957	D. I-I. Warnke, 1969
Lester E. Hill, 1958	James D. Parker, Jr., 1970
D. B. Paquin, 1959	Edward O. Lineback, 1971
L. W. Montgomery, 1960	Herbert B. Safford, 1972
Alfonso G. Pia, 1961	James Sutton, 1973
Elmer F. Judge, 1962	Edward J. Palmer, 1974
Robert S. Gilmore, 1963	J. Ralph Leatherman, 1975

Organizational Structure. Individual chapters are organized geographically into 13 regions, each of which holds business meetings, conducts regional conferences and educational seminars, and carries on various types of interchapter educational activities. Governing authority is vested in the International Board of Directors, which consists of one representative from each chapter. An annual meeting of the Board is held in conjunction with the International Data Processing Conference & Business Exposition sponsored by the Association. International directors, appointed by chapters, also represent their chapters at regional meetings.

Implementation of policy established by the Board is carried out by an Executive Council consisting of 21 members: President, Executive Vice-President, Secretary-Treasurer, Immediate Past President, four International Vice-Presidents (with the following areas of responsibility: Planning and Policy, Education, Certification and Testing, and International Affairs and Inter-Association Liaison) and 13 regional Vice-Presidents. Assisting the Executive Council in managing association affairs is a Corporate Operations Committee and a Policy and Planning Committee made up of members chosen from the Executive Council.

The local chapter is the heart of the Association. Every member must belong to a chapter, except those applying for an individual international membership, which is granted to qualified individuals living outside North America upon approval by the International Executive Vice-President. Extensive educational programs are carried on by the local chapters through regular monthly meetings, seminars, and other activities.

Regular membership is granted by the individual chapter Board of Directors to persons engaged as (1) managerial or supervisory personnel in EDP installations; (2) systems and methods analysts,

research specialists, and computer programmers employed in executive, administrative, or consulting capacities; (3) staff, managers, educators, and executive personnel with a direct interest in data processing; and (4) holders of the Certificate in Data Processing (CDP). Other types of membership are affiliate, fellow, and honorary.

A computer-equipped international headquarters with modern facilities, located in Park Ridge, Illinois, serves as the administrative nucleus of the Association. It provides a wide range of programs and services to local chapters and contributes to regional educational programs. Major departments are Membership, Research and Professional Services, Conferences and Communications, and Data Processing.

Programs and Services. DPMA members attend meetings, seminars, and conferences at the local chapter, and at regional and international levels. A major educational event is the Annual DPMA International Data Processing Conference & Business Exposition, attended by members and non-members from all parts of the United States, Canada, and other countries.

The Association was the first to introduce (in 1962) a certification program for computer management personnel. The Certificate in Data Processing (CDP) examination program is dedicated to the advancement of data processing and information management and to this end has established high standards based on a broad educational framework and practical knowledge. In 1970, DPMA also introduced the Registered Business Programmer examination, which seeks to identify those reaching the level of senior business programmers. Both examinations were developed by the DPMA Certification Council and are given annually in test centers at colleges and universities in the United States and in Canada. In 1974 DPMA transferred ownership of these examinations to the Institute for Certification of Computer Professionals (ICCP). Other programs offered to the membership include the Business and Management Principles one-day seminar, the video tape Management Development seminar, and Educator's Night for improving communications with the education community. DPMA encourages and provides assistance to student organizations interested in data processing in colleges and universities. It also offers the Future Data Processors Program for high school students, and provides counseling aid for Boy Scouts seeking the computer merit badge.

Other programs are being constantly developed to keep the membership abreast of changing devel-

opments in effective EDP management techniques and in technological advances.

Among DPMA publications are the monthly Data Management magazine (included in membership dues); *Guidelines to Data Processing Management*; *An Executive Briefing on the Control of Computers*; *Automatic Data Processing-Principles and Procedures*, and several data processing briefings for the student.

Its audiovisual program includes films and slide presentations ranging from introductions to data processing for the layman to general management subjects. In 1969, DPMA originated the Computer-Science-Man-of-the-Year Citation which in that year was presented to Commander Grace Murray Hopper, USNR. Subsequent recipients have been Dr. Frederick Phillips Brooks, Jr., 1970; Robert C. Cheek, 1972; Dr. Carl Hammer, 1973; and Prof. Edward L. Glaser, 1974.

I. L. AUERBACH

DATA SECURITY

For articles on related subjects see **COMPUTERS AND SOCIETY**; **RELIABILITY AND FAULT TOLERANCE**; and **SECURITY OF COMPUTER INSTALLATIONS, PHYSICAL**.

For articles on related terms see **HARD COPY**; and **RECORD**.

The protection of data against the deliberate or accidental access of unauthorized persons is rapidly becoming a major problem. Ultimately, the security of data depends on some combination of "locks," or access-control measures, for which certain users possess the "keys". No such combination is completely secure. For the intruder, the effectiveness of security measures is really only a matter of the cost of breaking the combination of locks as compared to the value (to him) of obtaining data in this way. Conversely, for someone wishing to maintain the security of data, the cost of devising and implementing a combination of locks on the data must be small relative to the cost of a breach of security.

In the case of, for example, military intelligence data banks, the information contained in them is considered to be of such value that almost no cost is spared to insure data security. Such systems, however, are clearly exceptional. This article deals instead with commercial or public data banks where

there are clear limits to the number of high-cost security measures that can be justified.

It is to be noted that in a computer system, the protection of the data itself and of software search and retrieval programs are treated almost entirely in the same manner; thus, the safeguards that apply to program security also apply to data security.

Classification of Degree of Confidentiality. In this section the term "user" describes a single person or a group of persons, all of whom have equal rights with respect to accessing a particular body of data and who have a common identity to the system. Three classes of data are defined for an automated system: public, limited-access, and private.

TYPE 1. PUBLIC DATA. Public data is open to all users, and no security measures are necessary as far as reading is concerned. When access is restricted to reading of the data, as it should be where data must remain unchanged, writing should be prevented. If it is not possible to prevent writing, check sums (a simple total of all data items) that should remain constant can be kept with data, and the data can be refreshed from a secure copy whenever a test total of the data does not agree with the check sum. If users are permitted to alter data, a lock must be maintained on the system to insure that while one user is making a change, no other user is permitted access to the data, since normally one user's alterations must be completed before another may begin.

TYPE 2. LIMITED-ACCESS DATA. Only authorized users have access to data of this type. This means that an authorization table must be kept in the system, indicating for each body of data the identity of all users with access rights. When a user requests access: (1) his identity should be authenticated, for example, by personal identification or password; (2) the authorization table should be checked to see that he has appropriate access rights; (3) a record in a log should be made of the event. The purpose of the log is to provide an audit trail or record that can be consulted whenever any trouble is suspected. All unsuccessful attempts to access data should be logged in order to provide an indication of a possible security leak. If the frequency of unsuccessful entry is larger than normal error expectation warrants, an alarm should be generated.

TYPE 3. PRIVATE DATA. This data is open to a single user only. When access to data is requested, the identity of the user should be authenticated to verify the fact that he is the owner of the data. Here again, a record of all unsuccessful attempts at entry should be logged.

DATA SECURITY

Access Rights to Data. Data that is not a program is usually organized into discrete files. A *file* is composed of a number of records, or factual statements, each relating to a particular thing; or, in a file containing personal data, each is related to a particular individual. A *record*, in turn, is subdivided into fields. A *field* is a precisely defined location within a record where information may be recorded.

In a file of personal data, certain fields enable the reader of the record to identify the person. Access to a file of personal information is often permitted on the basis of "need to know," and access to a particular record in a file is allowed on the basis of an explicit or implicit consent of the individual to whom the record pertains. It would therefore follow that if a person having access to a record needs to know only the information in certain fields of the record, he should not have access to other fields in the same record. For example, persons who are preparing statistical summaries from files do not need to know the identity of the person to whom each record applies, and therefore should not have access to identifying fields.

Frequently, persons having access to a file have access to *all* fields of *all* records. In a manual file in which records are maintained in a manila folder, it is difficult to arrange to do otherwise. In a computerized system, however, access can be permitted to the entire file, or can be restricted to certain records or to certain fields of the file.

Access rights might be defined as follows: read an item (e.g., file, record, or field); write an item so as to produce a change, either by adding a new item or by changing an existing item; delete an item.

The access rights of a user must be explicitly denoted in any situation where partial rights exist, such as a limited access file, or where reading is permitted but changes and deletions are not. It is possible to have a table or matrix stored with the data (or separately) which lists authorized users of the data and their access rights. Access to this table must be strictly limited to persons authorized to modify the table, usually only the owner of the data. In many cases, access control is assigned to the system itself, since in most computer systems the operations pertaining to the read or write functions are already under system control.

Physical Storage of Data. Data in an automated system can exist in many physical forms. Storage media may be classified into five categories: hard copy; display devices; magnetic tape and mountable disks; mounted magnetic tapes, disks, and drums; and magnetic core store.

HARD COPY. This is a term used for recording data that is more or less permanent and that can be stored, read, or written by humans independently of the computer hardware. Included in this category are printed pages, punched paper tape, punched cards, and microfilm. The security of hard copy is similar to the conventional security associated with manual files. The interpretation of the data by an intruder is usually very simple. Also, the destruction of the data requires the destruction of the medium. Machines for shredding hard copy are available.

DISPLAY DEVICES. These are devices on which data may be exhibited to a user but on which it has an evanescent form. As soon as it is no longer required, it will disappear. An example of this form of storage is a cathode-ray tube display. If such devices display sensitive data, they may to have be used in secure rooms where unwanted cameras or persons cannot observe them. When electric circuits are arranged to display images on one cathode-ray tube, stray electromagnetic radiation from these circuits might be amplified to produce a similar display on another such device that has no connection to the first.

Display devices like printers or card readers should also be appropriately shielded to guard against the possibility of electromagnetic eavesdropping. There is the possibility of telephone instruments acting as pickup devices for such radiations even when on the hook.

MAGNETIC TAPE AND MOUNTABLE DISKS. These are media on which data can be recorded as variations in magnetization. They can be erased and used repeatedly, although-as with most erasing processes-small traces of previously recorded information may persist. When the tapes or disks are not mounted on a computer device to read them, they cannot be read, but they can be erased or destroyed, for example, by strong magnets or fire.

A careful banking system in secure rooms under strict control must be maintained to prevent loss or violation of security during off-line storage. When tapes or disks are mounted on read/write devices, they become identical in nature to those integrated in the system (see below) of physical storage; when **unmounted**, they are similar in nature to hard copy.

MOUNTED MAGNETIC TAPES, DISKS, AND DRUMS. These mounted media are integral part\$ of the on-line storage system of a computer. They are usually classed as the secondary store, since the time to access information stored on them is long compared with the basic operation rate of the computer. Usually, the time for reading from or writing on them is overlapped by other operations. This means

that the individual user does not direct the reading or writing himself, but has to go through the intermediary of the computer operating system. Access control almost always resides in the operating system.

MAGNETIC CORE STORE. This category is the main first-level store of most computers. To operate, a program must be in the core store; to be acted upon, data must be in the core store. Thus, in the final analysis, on-line access to data must first be controlled by controlling access in the core store. Since all users are permitted to use core store, one after another, it is important to erase any residual sensitive data before allowing the next user's program to have control. Some operating systems do this clean-up job automatically; others do not.

Protection of Data in Core Storage. Each user in a multiprogrammed computer system is assigned, at a given instant of time, a region of the core store as his own private domain during execution of his program. The right to read from or write into this area of the store is protected by a key (usually through a hardware device). A directory of keys is kept in a table in the core area assigned to the operating system itself. This means that the user cannot alter the directory entry pertaining to his own core area. If he could, it might permit him to access some other user's core area. Each user core area is thus private to an individual user.

Sharing data and programs can be achieved by having an area of core that is common to the users concerned. If only specified users may share the data in this area, an authorization table must be maintained. Often it is sufficient to declare the area as "public" in order to insure that no access control is necessary.

Protection of Data in Secondary Storage. In a secure system, all requests to read and write on secondary storage must pass through the input/output control of the operating system. In order to issue a **READ** or **WRITE** instruction to a file in secondary storage, it is necessary for a user to alert the operating system that he intends to perform operations on the file by issuing an instruction. At this time, his access rights to the file are examined.

Private files are usually labeled with the name and system identification number of the user, and may even contain in their labels a password that must be matched against one provided by the user at the time of issuing the access request. Only the owner of a limited-access file should be permitted to

change the password. When a file has limited access, the access information is frequently stored separately from the data.

Protection of Data in Transmission. Wiretapping or electromagnetic eavesdropping is a security threat whenever data travels through the air or over wires that are not in a secure area. Many systems use common carrier facilities, and this presents many problems. Sensitive data that is to be transmitted from one location to another should be transformed (i.e., encrypted) to make it private. Privacy transformations that involve static methods of coding require a certain amount of work to break, but can usually be decoded after some effort.

The best coding techniques involve keys that are as long as the data to be encrypted. The string of characters for the key is generated from a basic starting number, just as a sequence of pseudo-random numbers can be generated. The same starting value yields the same sequence every time. It is nearly impossible to determine the starting value and the generating algorithm from eavesdropping on the transmission. The work required to break the code is very extensive.

Protection of Data Off-Line. Stored data in the form of hard copy, or on magnetic tape or removable disks, must be kept in a strictly controlled environment. Protection against accidental or wilful damage or theft must be insured. Access to the data will be basically through a manual system managed by a person charged with its security. There should be a record kept of all deposits and withdrawals from the data bank, after assuring that the person making the transaction has the appropriate access rights.

Frequently, data banks are located near the main computer installation where tapes are requested frequently, but it is common also for systems to have a separate repository of tapes containing data vital to regenerating the system. A remote storage vault in a protected location is essential for basic business or industrial data.

Integrity of Hardware. The fashion of having computers prominently displayed to the public is dying out. The need for precise environmental control has always meant that hardware was housed in special rooms, but it is increasingly apparent that protection and control of access to the rooms is also critical to security. Without this isolation, not only could the equipment be destroyed, but data also

DATA SET

could be compromised as it is being printed or displayed.

All persons having access to the rooms where hardware is kept should be properly identified and their "need to be present" should be verified. Systems of identification badges are common. Sometimes access is controlled by a security officer; sometimes by locks opened by badges or by combinations. The advantage of locks operated by badges or push-button combinations is that the combination can be more easily changed than can locks operated with ordinary keys. Thus, if there is any suspicion of a compromise of a lock, the combination should be altered. When the key is a badge, the rightful owner may have his picture displayed on it for further identification.

Where a piece of hardware is attached to the main computer hardware through a remote connection, the terminal equipment is often under minimal or no surveillance. As a result, at the present time, highly sensitive data is rarely handled in a computer system with remote terminals. It is important that remote users be properly identified and that the terminals be properly identified, not only at the time of beginning a "conversation," but also from time to time during any extended interaction. This is rarely the case except in military systems. No doubt the security of data in systems with remote terminals will be improved.

Integrity of Software. We have indicated that the security of data within the computer depends on the operating system. Many existing systems are complex; to some extent, their complexity protects them from invasion. However, accidental access routes (trapdoors) into the system have been found. When access routes via trapdoors are found, the usual result is that the system becomes inoperative; many of the breakdowns that occur daily in systems are the result of accidental entry into the operating system by an unsuspecting user.

To be secure, operating system structure should be cleanly designed and the documentation openly available. Secrecy should not be a requirement for a secure system. Perhaps only critical parts of operating systems need be under strict security control (e.g., tables of access rights and the programs that validate these rights) to insure data security.

REFERENCES

1969. Hoffman, Lance J. "Computers and Privacy:

A Survey," *Computing Reviews*, Vol. 1, No. 2 (June).

1970. Friedman, T. D. "The Authorization Problem in Shared Files," *IBM Systems Journal*, Vol. 4, No. 4.

J. N. P. HUME

DATA SET

For articles on related subjects see **ACCESS METHODS**; and **FILES**.

This article deals with the software meaning of the term "data set"—a collection of related data items (Clarke, 1966)—and *not* the hardware meaning, which defines "data set" as a device used to couple a computer data link to a telephone line. Examples of data sets within our context are the collection of student records within a university, the collection of inventory items in a warehouse, the set of records describing books in a library, etc. The major objectives of grouping data items into data sets are efficient retrieval, searching, sorting, and recognition.

A data set may be described by a *data set label*, which might contain the name of the data set, its boundaries in physical storage, and certain characteristics of the data items within the set. Within a unit of physical storage (or *volume*), the set of all data set labels may be considered as a "table of contents." Searches, for example, may be expedited by searching the "table of contents" for data set labels exhibiting desired characteristics, and then searching for data items with more specific characteristics among only those data sets whose labels passed the first search.

To enhance the operating characteristics of the storage media used, data sets may be organized in various structures:

1. *Sequential*, as on magnetic tape.
2. *Indexed sequential*, where data items are stored sequentially on a key, but are also accessible via index tables maintained by the system.
3. *Direct*, without index tables accessing of data items is up to the programmer.
4. *Partitioned*, where sequentially organized data

sets are divided into members, and a directory is maintained of members' names and their storage locations.

5. *Telecommunication*, where the data items in the set are messages organized into queues (e.g., for communication between a computer and a remote terminal).

REFERENCE

1966. Clark, W. A. "Data Management," *IBM Systems Journal*, Vol. 5, No. 1.

S. C. BREWER

DATA STRUCTURES

For articles on related subjects see **LISTS AND LIST PROCESSING; STACK; STRING; and TREE.**

For articles on related terms see **DATA BASE AND DATA BASE MANAGEMENT; GRAPH THEORY; and QUEUEING THEORY.**

The term "structure" is used in many different fields to denote objects that are constructed in a regular and characteristic way from their components. A data structure is a structure whose components are data objects.

EXAMPLE. The arithmetic expression $3 + 4 * 5$ is constructed in a systematic way from data components that are integers such as 3, 4, 5, and operators such as + and *. The structure of this expression may be thought of as either a string or a tree structure in which each operator is the root of a subtree whose descendants are operands (Fig. 1).

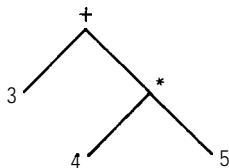


Fig. 1. A tree structure.

When this data structure is stored in a computer, it must be stored so that components are

readily accessible. This may be done by storing the expression $3 + 4 * 5$ as a character string A so that the *i*th character is retrieved by referring to the element A [*i*] or A (*i*), with the use of brackets or parentheses depending upon the programming language being used. Alternatively, the string may be stored as a list structure, in which the vertex associated with + has a left son 3 and a right son *, which in turn has left and right sons 4 and 5 (Fig. 2).

Figs. 1 and 2 illustrate the relation between data structures, which specify *logical* relations between data components, and storage structures, which specify how such relations may be realized in a digital computer. The storage structure of Fig. 2 could be represented in a digital computer by five three-component storage cells, where each cell has one component containing an operator and two components respectively containing a pointer to the left and right sons. The three cells that have no successors contain special markers in their pointer fields, here indicated by the word "nil."

In order to define a class of data objects having a common data structure, it is usual to start with a class of primitive data elements called "atoms," or elementary objects, and to specify *construction operators* by means of which *composite objects* may be constructed from the atoms. In the preceding arithmetic-expression example, the atoms are operands (integers) and arithmetic operators. The construction operators specify how expressions are built up from operators and operands. The set of construction rules that specify how operators are built up from operands is sometimes referred to as a "grammar."

In order to access and manipulate composite objects specified by a given set of atoms and construction rules, selectors must be defined which allow components of a data object to be accessed, and *creation* and *deletion* operators must be defined which allow components of data structures to be created and deleted. Data structures may be characterized by the nature of their accessing and their creation and deletion operators.

Some of the basic terminology relating to data structures will be mentioned by considering commonly occurring data structures such as arrays, lists, trees, stacks, and queues.

An *array* is a data structure whose elements may be selected by integer selectors called "indexes." If A is a one-dimensional-array data structure, then A [3] or A (3) refers to the third element of A. If B is a three-dimensional array, then B[I,J,K] or B(I,J,K) refers to the I,J,K element (b_{ijk}) of the array B. The set of all elements of an array are generally created and deleted at the same time by means of *decla-*

DATA STRUCTURES

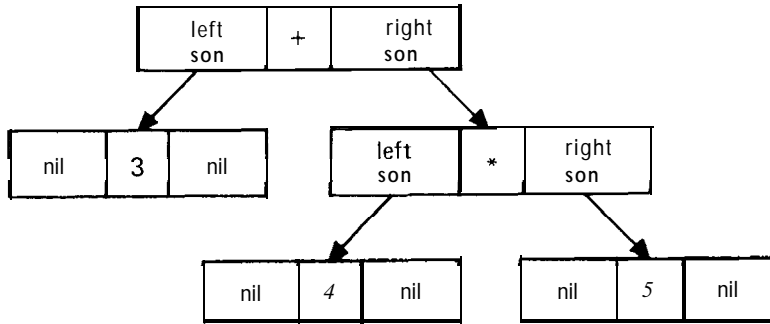


Fig. 2. Storage structure for the tree structure of Fig. 1.

rations, as illustrated by the following examples:

`DIMENSION A (1,100)` Fortran array declaration
`integer array A[1:N];` Algol 60 array declaration
`[1:N] int A;` Algol 68 array declaration

In Fortran, the declaration "`DIMENSION A(100)`" serves to reserve a block of cells for the array A at compile time. In Algol 60 or 68, declarations create an instance of the declared data structure when they are executed. Thus, execution of the declaration "`integer array A[1:N]`" causes allocation of a block of N storage cells large enough to hold integers using the current value assigned to the variable N , and activates an accessing mechanism so that $A[i]$ will refer to the i th allocated cell.

The arrays introduced above are homogeneous because all elements of an array have the same data type, and are rectangular because all vectors in a given dimension have the same size. Programming languages such as Cobol and PL/I permit non-homogeneous, nonrectangular arrays to be declared. The following is a PL/I declaration of a PAYROLL record with a 50-character name field, fields of the mode FIXED for the number of regular and overtime hours worked, and a field of the mode FLOAT for the rate of pay:

```
DECLARE 1 PAYROLL
  2 NAME CHARACTER(50),
  2 HOURS
    3 REGULAR FIXED,
    3 OVERTIME FIXED,
  2 RATE FLOAT;
```

If it is desired to refer to the number of overtime hours in the record PAYROLL, then this is given by

PAYROLL.HOURS.OVERTIME. That is, component names rather than indexes are used to access a given element of the data structure.

List structures, just as array structures, may be characterized by their accessing creation and deletion operators. Elements of a list structure are generally accessed by "walking" along pointer chains, starting at the head of the list. In a linear list, each list element has a unique successor and the last element has an "empty" successor field, usually denoted by the symbol "nil." In general, list elements may have more than one successor, and lists may be circular in the sense that pointer chains may form cycles. Knuth (1968) introduces doubly linked lists that have forward and backward pointer chains passing through each element, and a number of other kinds of lists. Fig. 3 illustrates a doubly linked circular list named L whose head element H is linked both to the next element A and to the last element B .

If the forward pointer is referred to by $RLINK$ (for right link) and the backward pointer is referred to by $LLINK$ (left link), then the second list element (labeled A) may be accessed in either of the two following ways:

$RLINK(L)$ Forward chaining
 $LLINK(LLINK(L))$ Backward chaining

Insertion- and deletion of elements in a list is accomplished by creation of a new list cell and by updating pointers of existing list elements and the newly created list element. Fig. 4 illustrates that the insertion of the list element X between the list elements A and B requires updating of the $RLINK$ of A , the $LLINK$ of B , and initialization of the R and L links of X .

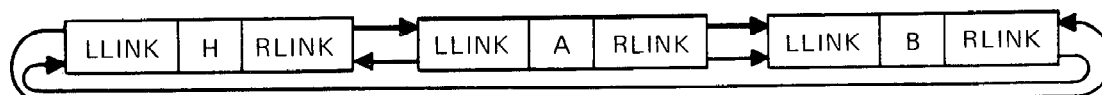


Fig. 3. Doubly linked circular list L .

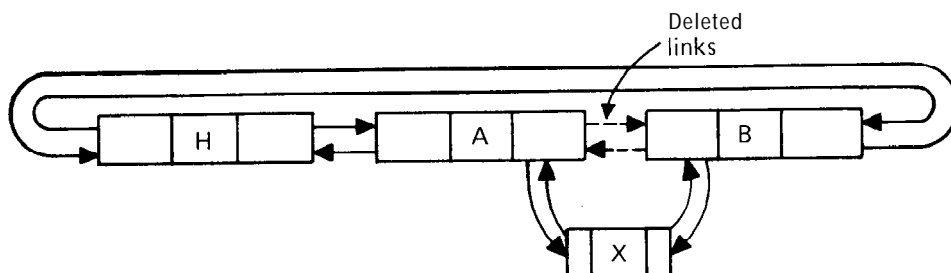


Fig. 4. Insertion of X in Fig. 3.

The instructions to perform this insertion might be as follows:

```
create X
RLINK(A) = L(X) (L(X) = Location of X)
LLINK(B) = L(X)
RLINK(X) = L(B)
LLINK(X) = L(A)
```

The list processing language Lisp, which was developed by John McCarthy in the late 1950s, is probably the most important list processing language. The list format and instruction repertoire of Lisp will be briefly illustrated. For ease of presentation, however, we will use a notation different from that actually used in Lisp.

List elements in Lisp have two components selectable by the selectors *first* and *rest*. If L is a list then *first*(L) selects the first element of the list, which may be either an atom or a sublist, and *rest*(L) selects the rest of the list. The list $((A, B), C)$ is represented in Lisp by the list structure of Fig. 5.

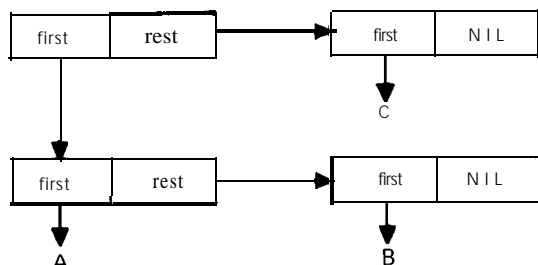


Fig. 5. Representation of a list L .

For $L = ((A, B), C)$, *first*(L) = (A, B) , *rest*(L) = (C) , *first*(*first*(L)) = A and *rest*(*rest*(L)) = NIL .

Lisp also has a construction operator *cons*[$X; Y$] which constructs a list L such that *first*(L) = X and *rest*(L) = Y , and a predicate *atom*(X), which is true when X is an atom and false otherwise. In the above example, *atom*(*first*(L)) = *false* since *first*(L) = (A, B) , but *atom*(*first*(*first*(L))) = *true*.

In general, any language for the manipulation of data structures has not only *selectors* for selecting components of a data structure but also *constructors* for constructing data structures from their components, and *predicates* for testing whether a given data object has certain attributes. Lisp illustrates particularly clearly the role of selectors, constructors, and predicates in a programming language.

List structures are a flexible storage structure for objects of variable sizes or tables of fixed-size objects in which insertion and deletion is frequently required. A number of special classes of list structures will now be considered in greater detail.

A *tree* is a list in which there is one element called the "root" with no predecessor and in which every other element has a unique predecessor. That is, a tree is a list that contains no circular lists, and in which no two list elements may have a common sublist as a successor. Elements of a tree which have no successor are called "leaves" of the tree. Tree elements, just as list elements, are generally accessed by walking along a pointer chain. However, the guarantee that there are no cycles or common sublists makes it possible to define orderly procedures for insertion and deletion of subtrees.

A *stack* is a linear list in which elements are accessed, created, and deleted in a last-in-first-out

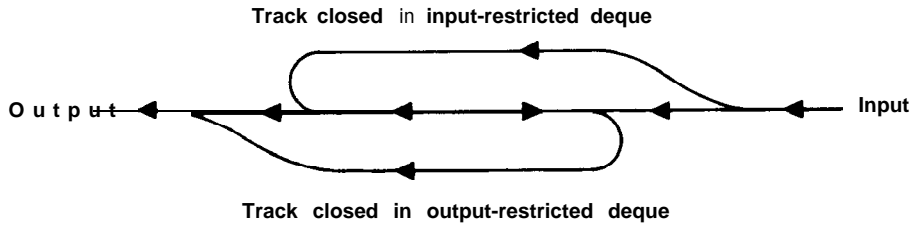


Fig. 6. A double-ended queue (deque).

(LIFO) order. In order to access an element in a stack, it is necessary to delete all more recently entered elements from the stack. Thus, only the top of the stack is accessible. The two principal stack operations are *poping* and pushing. If S is a stack, then $pop(S)$ causes the top element of the stack to be deleted and $push(S, x)$ causes x to be placed on top of the stack.

A queue is a linear list in which elements are created and deleted in a first-in-first-out order. A line of people waiting to be served in a cafeteria is a queue, since the person having waited longest is always the first to be served (deleted from the queue). In contrast, employees in a large organization generally form a stack with regard to being fired.

A generalization of queues and stacks in which elements may be added and deleted at both ends of a linear list is called a “deque.” A deque is said to be input-restricted if input is possible at only one end, but deletion may occur at both ends. A deque is said to be output-restricted if output may occur at only one end, but input may occur at both ends. Fig. 6 illustrates by means of a railway-switching network the notion of a deque with input and/or output restrictions (see Knuth, 1968, p. 236).

Data structures include numerical structures such as integers that have arithmetic operations applicable to them, and nonnumerical structures such as arrays, list, and trees whose primary purpose is to keep track of relations among data objects rather than to manipulate them.

Computational structures may be studied and analyzed at many different levels of abstraction. We have already remarked on the difference between logical data structures and the storage structures in terms of which they are realized. The characterization of structure by logical relations among components is clearly more abstract than the realization of the logical structure by particular configurations of cells and pointers. It is convenient to introduce an additional higher-level mathematical level of abstraction in which logical relations among compo-

nents of a data structure are characterized even more abstractly by mathematical relations, and an additional lower-level “hardware” level of abstraction that specifies how storage structures are realized at the hardware level.

1. *Mathematical structure* is defined by specifying a set of objects and a set of operators (functions, relations) for transforming objects into other objects.

2. Data *structure* is defined by labeled-directed graphs that allow characteristic operators on data objects having the given structure to be naturally and simply defined by means of graph transformation rules. A given mathematical structure may, in general, be represented in many different ways by a data structure.

3. *Storage structure* is defined by storage cells with pointers between storage cells. Storage structures, like data structures, are chosen so that operators applicable to computational objects represented by a given storage structure may be simply and efficiently defined. There are, in general, many different storage structures that realize a given data structure.

4. *Hardware structure* specifies how storage structures and transformations of storage structures may be realized at the hardware level.

EXAMPLE. In modeling data bases, the mathematical level of abstraction models data bases as mathematical relations, the data structure level considers data bases to be directed labeled graphs, the storage structure level considers how the directed graphs representing particular data configurations can be efficiently realized by storage structures, and the hardware structure level considers hardware and microprograms for realizing particular ‘storage structures.’

Although these four levels of *structure* specification are somewhat arbitrary, they appear to be “robust” in the sense that attempts to quantify the notion of abstraction invariably result in something

similar to the above characterization. For example, in considering abstraction for program structure, we generally distinguish between mathematical structure, program structure, implementation structure, and hardware realization. These distinctions are very similar to the previously discussed distinctions for the data structure case.

Data structures capture the notion of computational structure at a level that is sufficiently abstract to emphasize logical relations among components of a data object, independently of details of implementation but at the same time sufficiently concrete to preserve some relation between a structure and its computational realization. Data structures thus represent an appropriate and practicable level of abstraction for characterizing computational structure, and it is for this reason that the study of data structures is important in computer science.

REFERENCE

1968. Knuth, D. E. *The Art Of Computer Programming*, vol. 1. Reading, Mass.: Addison-Wesley.

P. WEGNER

DATA STRUCTURES, SET CONCEPTS FOR

For articles on related subjects see **ACCESS METHODS; DATA BASE AND DATA BASE MANAGEMENT; DATA STRUCTURES; FILES;** and **STORAGE-MANAGEMENT STRUCTURES.** For articles on related terms see **LIST-PROCESSING LANGUAGES; RECORD;** and **SIMULATION.**

The data base set concept unifies several programming techniques (table, list, chain, ring, file, and field array) that have been in common usage for most of the history of computers. (This concept is a specialization of the more general mathematical set concept from which the data structure set gets its name and many of its properties.) In this paper the word "set" will always be used in the data structure and not the mathematical sense.

Many software products support the set concept. The list-processing languages, such as IPL-V and Lisp, have used the set concept to support the organization of program structure. Simulation lan-

guages, such as Simscript and Simula, have used set concepts to assist in modeling the subject of study. In the data base management area, the Honeywell integrated data store (IDS) system (Bachman and Williams, 1964) pioneered broad usage of the set concept to process complex manufacturing and banking problems. IDS uses the chain form of set implementation. General Motors Research (Dodd, 1966) produced a similar system, Associative Processing Language (APL) for graphic display purposes. After six years of study, the CODASYL Data Base Task Group (ACM, 1971) produced a specification that is being integrated into the Cobol language. The Cobol Data Description and Data Manipulation Language extensions make available the set description and manipulation capabilities of the IDS and APL systems. IBM's Information Management System (IMS II) and Informatic's MARK IV support hierarchical set structures with the multiple-level record array technique. Cobol and PL/I's recognition of the set concept is limited to constructing sets of member records with field arrays.

Record, Field, and Set Concepts. The set is one of three complementary concepts (record, field, and set) needed to build and store data structures that closely approximate their natural world counterparts. If the natural world is considered in terms of the entities that exist, the attributes that describe them, and the relationships that associate them, then the equivalent information system concepts are record, field, and set, respectively.

In a simple example taken from a school situation, the entities would be the teachers and the children. Some of the attributes of a teacher are "name," "grade level," and "classroom." Some of the attributes of a child would be "name," "age," and "parent name." A relationship exists between teachers and children. In an information system model of this natural world situation, two classes of records (one for teachers, one for children) would be created. In each teacher record there would be a field to store the teacher's name, another for the grade level, and another for the classroom number. Each child's record would have a field for the child's name, another for his age, and yet another for his parent's name. The information system could tie each child's record to his teacher's record in one of the several ways that have been invented to implement the set concept. This might be done by physically placing all the child records after their teacher's record array in the file. This is called a "table" or "record array."

DATA STRUCTURES, SET CONCEPTS FOR

The **data** structure set concept thus described is a refinement of the mathematical set concept; i.e., in the data structure set, the set definition is embodied in the instance of the “owner” role. Set membership is embodied in the instance of the “member” role. Records may concurrently have many roles as owner and member of different sets. This property permits the creation and manipulation of complex structures that model the complexity of the real world. In this refinement of the mathematical set concept, one can go reversibly either from owner as definition to members or from any member to owner to reestablish the set definition.

For data structure sets, the set definition is normally based upon the value of some field or fields within the owner record, while the membership in the set is established by the matching value of an equivalent field or fields within a potential member record. Advantage is frequently taken of this phenomenon by removing the fields from the member records that carry the matching data and depending upon the owner record for reconstruction.

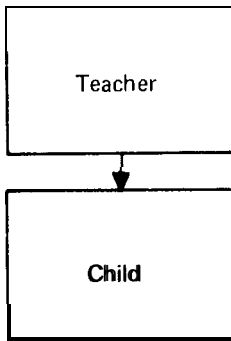


Fig. 1. Teacher to child relationship.

A data structure diagram (Bachman, 1969) would illustrate the teacher/child structure as shown in Fig. 1. It uses a box to represent the entity class concept and an arrow to represent the **simple relationship** class concept. (A simple relationship is a $1:n$ relationship between entities. Alternately, a complex relationship is an $m:n$ relationship.) In this case there are two boxes, one for all the teacher entities and one for all the child entities. The arrow symbolizes the relationship that each teacher may have: zero, one, two, or more children. However, each child has only one teacher.

In the school example above, we said that a teacher has the role of “owner” of a teacher/children

set. To extend this example, we will recognize that in most schools the relationship between teacher and child is not a simple relationship ($1:n$), but is rather a complex relationship ($m:n$), since the children have different teachers for different subjects. This complex relationship of teacher:child may be transformed into a new relationship entity, “pupil,” and two simple relationships, teacher: pupil and child: pupil. The teacher has many children as pupils in her classes and, as a pupil, the child has many teachers. This new view is illustrated in Fig. 2. The new “pupil” entity has the attributes “subject” and “hour,” which serve to describe and differentiate one relationship entity from another. A child may have the same teacher for several subjects.

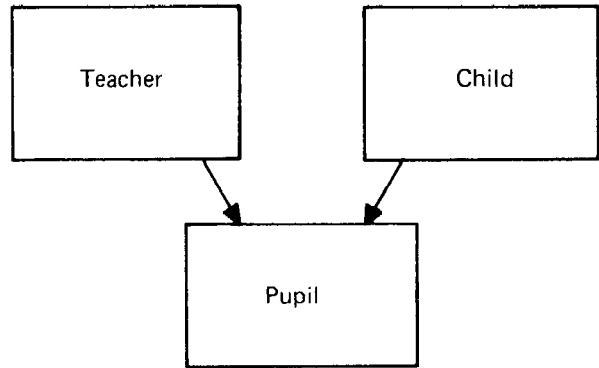


Fig. 2. Teacher/pupil/child relationships.

Set Formalisms. The data-structure set concept has four basic properties:

1. A set has one, only one, and always one record in the owner role (the teacher in Fig. 1 or Fig. 2).
2. A set has zero, one, or more records in the member role, and the number varies with time (the child in Fig. 1).
3. Any record may be the owner of zero, one, or more sets concurrently.
4. Any record may be a member in zero, one, or more sets concurrently, and thus be simultaneously owned by several owner records (the pupil in Fig. 2). Each record may appear only once as a member of a particular set. The member roles do not interfere with the owner roles.

Fig. 3 expresses the four basic properties of the set concept as a data structure diagram. The numbers used in the list above to enumerate the set properties

are shown in order to point out their effect upon the structure. Only slight additions are necessary to complete the data structure concepts that govern data processing.

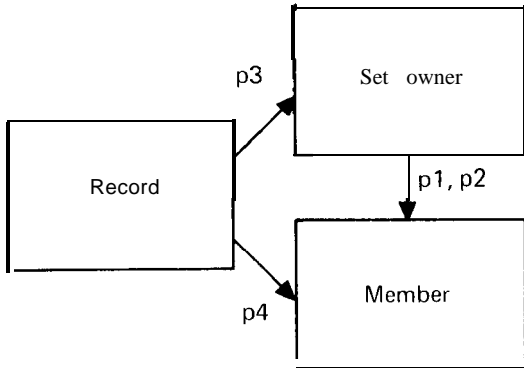


Fig. 3. Record/owner/member entity relationships.

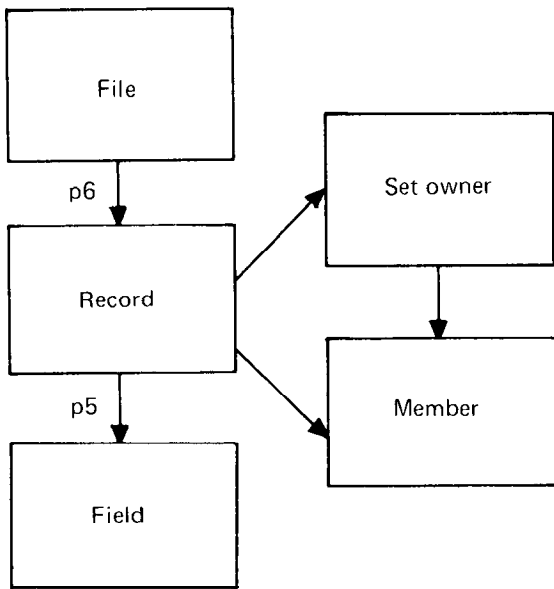


Fig. 4. Structure of file/record/field entity relationships

The fields of a record (p5 in Fig. 4) have been previously mentioned as the means of recording the attributes of an entity. All records must be stored in some container for safekeeping and reference. The file (p6) serves this function. Fig. 4 is an extension of Fig. 3, with the field concept and the file concept shown. The arrow from the box marked "file" to the

box marked "record" symbolizes the fact that a file may hold zero, one, or more records, but a record may be in only one file. The arrow from "record" to "field" symbolizes the fact that a record may have zero, one, or more fields, but a particular field may appear in only one record.

Set Ordering. The notion of "next" and "prior" are important concepts to procedural algorithms that are basic to problem solving in a stored program computer. In addition to the procedural limitation of handling one record at a time, there are important simplifying consequences to an algorithm if the member records within a set can be delivered to it in a predefined data-value ordered sequence or a time-of-insertion ordered sequence (FIFO or LIFO). The notions of "first" and "last" are vital to starting and stopping the iterative execution of these algorithms. Thus, the ordering of members in a set is a prerequisite to rational manipulation of the set.

Motivation of Set Concept. The primary motivation of associating records into sets within a file is to model the natural world relationships and to assist in the accessing of selected records within the file that represents some particular relationship. The set access methods fall in between and complement the more traditional access methods. They are listed in Table 1.

Table 1. Access methods.

Method	Use
Direct Access	Retrieves one record
Data-Key Access	Retrieves one record
Set-Owner Access	Retrieves one record
Set-Member Access	Used iteratively; retrieves each member of set
File-Sequential Access	Used iteratively; retrieves each record in file

The first four access methods in Table I are primarily used in transaction and inquiry processing, where there is a need to determine the recorded status of a particular entity or of a related group of entities, or to update their recorded status. The file-sequential access method is primarily used for periodic batch file updating and report generation. It is possible for the same record to be accessed by any of the five methods, as the occasion may require. Similarly, it is possible to use these access methods in combination to achieve a particular effect.

Table 2. Retrieval opportunities.

Given	Access Method	Determine
The owner	Set member	First member, or get empty-set notice
The owner	Set member	ith member, or get out-of-set notice
The owner	Set member	Last member, or get empty-set notice
Any member	Set member	Next member, or get last-of-set notice
Any member	Set member	Prior member, or get first-of-set notice
Any member	Set owner	Owner of set

Taking the example of Fig. 2, a teacher's record might be retrieved by the data-key access method and then her pupils' records could be retrieved by set-member access method. For each pupil record, the child's record may be retrieved with the set-owner access method. Alternately, retrieval might start with data-key access to the child's record and then proceed to access all pupil records of the child, and hence the teacher's records. The basic retrieval opportunities derived from a set are given in Table 2.

Operations on Sets. There is a family of primitive operations that apply to sets. These are complementary to the primitive operations on records and fields, which are better known, Table 3 gives the primitive operations on all three for comparison.

Table 3. Primitive operations on records, fields, and sets.

Object	Operation
Record	Create
	Access
	Destroy
	Test for record class name
Field (content)	Initialize
	Reference
	Alter
	Test for null-value status
Set	Insert member
	Remove member
	Access set owner (record)
	Access set member (record)
	Test for set emptiness status
	Test for member insertion status

The "insert" and "remove" operations on sets are the means by which a member record is procedurally introduced into a set and extracted. The insertion may be the first or last member, or logically between any two members, depending upon the set-ordering rules established for the set. The two set-access modes were described under "Motivation of Set Concept," and enumerated in Table 2. The

two set-test operations relate to whether or not a particular set is currently empty (owner record, but no member records) and whether or not a record is currently inserted as a member of a particular set. More elaborate operations exist in higher-level languages (Bachman and Williams, 1964; Dodd, 1966; CODASYL Task Group, 1971), but they are based upon these primitives.

Set Descriptions, Set Classes, and Set Occurrences. The sets described to this point have been completely free of any restrictions with regard to the class of record that could appear in either the owner or member role, or both. In Cobol, PL/I, and most other data processing languages, there exist the concepts of record description, record occurrence, and record class.

The record *description* is the "01" entry that appears in the Data Division of a Cobol source program. It is concerned with providing a record name and other attributes of the record.

A *record occurrence* is an instance of a record created in accordance with a record description.

A *record* class consists of all the records that have been, or will be, created in accordance with a particular record description.

In parallel with these record concepts, there are equivalent concepts for sets. A set description defines a set class name, set-owner selection criteria, set-member eligibility rules, and set-member ordering rules. A set occurrence is an instance of a set created in accordance with a set description. It is owned by a particular record, and it holds specific record occurrences as members. A set class includes all the set occurrences that have been or may be created in accordance with a particular set description.

The main purpose of both the record class and set class concepts is to create a strong organizing force. For example, it changes a data base with a million records from a million special situations into one where 40 or 50 situations exist: one situation for each record class and each set class. It changes the

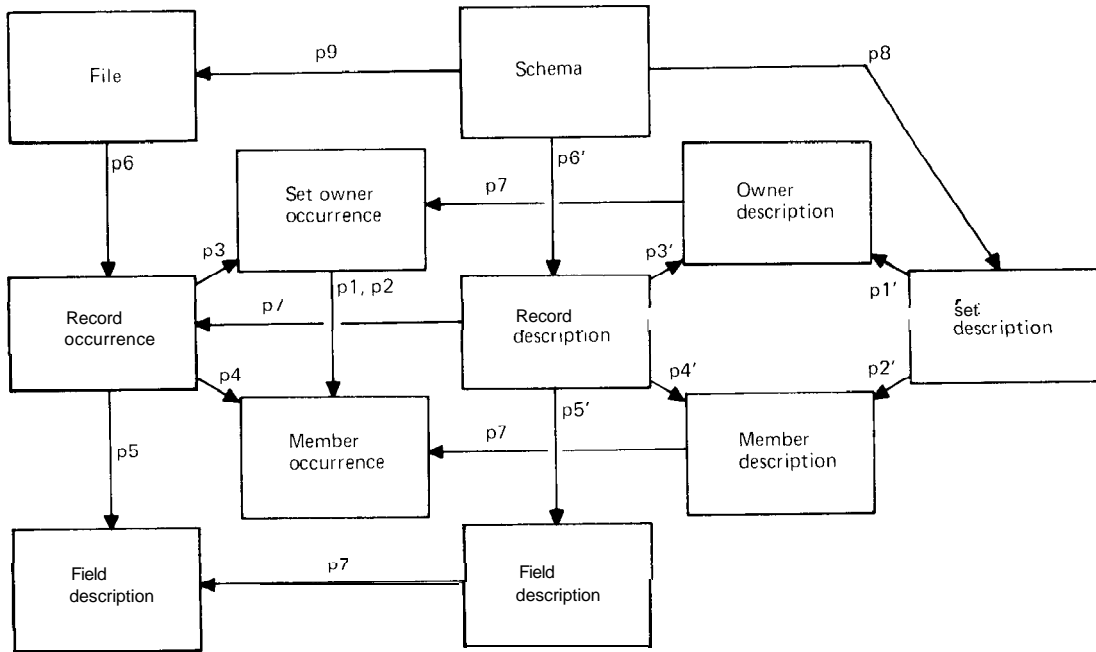


Fig. 5. Occurrence/description structures.

problem from something that could be chaos into something that is manageable.

The data structure diagram in Fig. 5 illustrates the integration of the description structure and occurrence structure, for data as illustrated in Fig. 4. There is a corresponding element in the description structure for each occurrence element in the occurrence structure. An arrow points from the descriptive element to its equivalent occurrence element, symbolizing the fact that there may be zero, one, or more occurrence elements for each descriptive element. These occurrence elements are the embodiment of the class property and are all labeled "p7."

The properties p1 through p6, previously described in Figs. 3 and 4, are illustrated on the occurrence structure, and their descriptive counterparts (p1', p2', . . . p6') are mapped onto the description structure. It should be noted that the "set-owner occurrence" block in the occurrence structure is controlled by two description boxes in the description structure. The "owner description" block describes the eligibility of an occurrence of a record class to serve in the owner role, whereas the "set description" block describes the set class as a whole. This separation is necessary because several record classes at the description level may be eligible to serve as set owners while, at the occurrence level, one and only one record is the actual owner of a set

occurrence. All the set descriptions in the schema are indicated by the arrow p8. All the files created in accordance with the schema are indicated by the arrow p9.

Auxiliary Uses of Sets. Assuming that the major usage of sets is to organize and provide access to records in an application data base, then all other usages within an information system are auxiliary to the primary purpose. The tabulation below lists areas of system software and enumerates for each some usages in its respective area of the set concept. This list is intended to be illustrative of obvious usages and is in no way complete.

1. Data base systems
 - (a) Index construction (index sequential and index random).
 - (b) Data description structures.
 - (c) Shared access control lists.
2. File systems
 - (a) Catalog construction.
 - (b) Access rights control.
3. Message systems
 - (a) Construction of mailbox indexes.
 - (b) Queueing messages.
 - (c) Accessing multielement messages.

DATA TYPE

4. Programming systems
 - (a) Controlling program libraries.
 - (b) Text editing.
 - (c) Program control structure.
 - (d) Symbol reference and symbol definition structures for binding.
 - (e) Intermediate program form for compilation.
5. Operating systems
 - (a) Queues of jobs.
 - (b) Resource allocation tables.
 - (c) Deadly embrace detection.
 - (d) Queues of processes waiting on events (I/O completion timer).
 - (e) Dispatching queues.

Summary. In summary, the set concept represents a new organizing force whose potential is just beginning to be realized. When fully recognized at the programming language level, in the training of programmers and system analysts, and in system software, it will substantially reduce some of today's problems, give clearer direction for file and application design, and improve the tenuous reliability normally associated with the development of new application systems.

REFERENCES

1964. Bachman, C. W., and S. B. Williams. "A General Purpose Programming System for Random Access Memories," Fall Joint Computer Conference.
1966. Dodd, G. G. "APL-A Language for Associative Data Handling in PL/I," Fall Joint Computer Conference.
1969. Bachman, C. W. "Data Structure Diagrams," Data Base (Quarterly News Letter of ACM-SIGBDP), Vol. 1, No. 2.
1971. ACM. "CODASYL COBOL Data Base Task Group Report" (April).

C.W. BACHMAN

DATA TABLET. See RAND TABLET.

DATA TYPE

For articles on related subjects see ARITHMETIC, COMPUTER; DATA STRUCTURES; and PROCEDURE-ORIENTED LANGUAGES.

For articles on related terms see BOOLEAN ALGEBRA; DECLARATIVE STATEMENT; and STRING.

A data type is an *interpretation* applied to a string of bits. Data types may be classified as structured or scalar. Scalar data types include real, integer, double precision, complex, logical (also called "boolean"), character, pointer, and label.

Structured data types are collections of individual data items of the same or different data types. An array is a data type that is a collection of data items of the same data type. Records, structures, or files are data types that are collections of data items of one or more data types.

Most programming languages provide a declaration statement or a standard convention to indicate the data type of the variable used. Thus, when the contents of the variable are accessed, they may be interpreted in the proper manner. This is necessary, since a string of bits may have several meanings, depending on the context in which it is used.

The *real* data type, which contains a normalized fraction (mantissa) and an exponent (characteristic), is used to represent floating-point data, usually decimal.

The *integer* data type is used to represent whole numbers, i.e., values without fractional parts.

Double precision is a generalization of the real data type, providing greater accuracy and sometimes a greater range of exponents.

Complex data contain two real fields representing the real and imaginary components of an imaginary number $a + bi$ (i is the square root of -1).

Logical data is of the true-false form; i.e., there are only two possible values, true or false.

Character data is the internal representation of printable characters. Some coding schemes (BCD) permit 64 characters and use six bits; others (EBCDIC and ASCII) permit up to 256 characters and use 8 bits.

Label data refers to locations in the program and pointer data refers to locations of other pieces of data.

The commonly used operators for addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (** or \uparrow) may be applied to real, integer, double precision, or complex data in higher-level language programs, with a few restrictions. The actual operation that takes place depends on the data type of the operands. Although some language processors permit "mixed mode" expressions (i.e., expressions involving operands of differing data

types), this is accomplished by converting the operands to a common data type before the operation is performed.

For example, to execute

$$N = (TEST + 90)/3,$$

the integer value 90 is converted to a real value, 90.0, so that it may be added to the value of `TEST` (assumed to be real-valued). Before the resultant real value can be divided, the integer value 3 must be converted to a real value, 3.0. Finally, the real result is truncated and converted to an integer so that it may be stored in the integer location `N`.

The logical operators and (`.AND. OR AND or &`), or (`.OR. OR OR OR }`), not (`.NOT. OR NOT OR —`), implies, and equivalence may be applied to logical data having true or false values only. Character operations include concatenation and selection of substrings. For all data types, the assignment operator (`=` or `←` or `:`) may be used to copy the contents of one location into another, and relational operators may be used to compare values of data items.

Certain programming languages (Snobol or Algol 68) are extendable in the sense that users may define new data types to suit the needs of a particular problem.

REFERENCE

1973. Wirth, Niklaus. *Systematic Programming: An Introduction* Englewood Cliffs, N.J.: Prentice-Hall.

B. SHNEIDERMAN

DEADLOCK

For articles on related subjects see `INTER-`
`RUPT`; and `MULTIPROGRAMMING`.

For article on related term see `TASK`.

A task (process) in a multiprogramming system is *deadlocked* if it cannot proceed because it is waiting for an event that will never occur. A system deadlock or a deadlock situation exists if one or more tasks in a system is deadlocked. An example given by Holt (1972) is the following PL/I program, which would cause a deadlock under OS/360:

```
REVENGE: PROCEDURE OPTIONS (MAIN, TASK);
          WAIT(EVENT);
END REVENGE;
```

The task associated with this program will wait forever unless it is somehow removed by the computer operator, since there is no provision for signaling that `EVENT` has occurred.

The events waited for by deadlocked tasks are often resource assignments. Suppose tasks `PETER` and `PAUL` both require simultaneous use of resources `CIRCLE` and `SQUARE` (which might, for example, each be a tape drive) in order to proceed. Assume that `PETER` has been assigned `CIRCLE` and `PAUL` holds `SQUARE`. Say `PETER` requests `SQUARE`, but must wait until it is released by `PAUL`. However, `PAUL` will release `SQUARE` only at task completion time, and in order to proceed to completion it is necessary for `PAUL` to have `CIRCLE`, which is held by `PETER`. The two tasks have requested resources in opposite order and have become involved in a *mutual* deadlock, or *circular wait*. More than two tasks may be involved: A situation may exist in which `PETER` is waiting for `PAUL` who is waiting for `FRED` who is waiting for `SAM` who is waiting for `PETER`. Indeed, all tasks in a multiprogramming system may be involved, in which case there is a *total* deadlock. In spooling systems, total deadlock might occur because of competition for spooling space on the disk. This would be the case if no task released disk space until it was completed and no task could complete without additional disk space. (Spooling space is not necessarily continually emptying in this system; it empties only when tasks are complete.)

Another term for deadlock (due to E. W. Dijkstra) is "deadly embrace," and still another is "knot." Some IBM logic manuals call a deadlock an "interlock."

Various ad hoc and systematic methods have been suggested for prevention of deadlocks, or for detection and subsequent recovery. These methods are reviewed in Holt (1972) and Coffman et al. (1971). Included in the ad hoc methods are conventions for requesting resources in specified order, and constraints on the amount of time a task is permitted to wait for an event. That these ad hoc techniques work fairly well is evidenced by the fact that deadlocks are not a serious problem in the operating systems of the mid-1970s. Ad hoc prevention methods are routinely used and deadlocks occur infrequently. Those deadlocks that do occur are resolved mostly by the computer operator, who may abort a deadlocked task, preempt resources from one

DEBUGGING

task in order to allow another task to continue, or restart the system.

It is possible, however, that the deadlock problem may be more pressing in the future as systems are developed with increased resource sharing and stronger concurrency. In this event, the systematic methods would appear more attractive, even though their cost might be relatively **high**. At present, most systematic methods confine themselves to those deadlocks attributable to contention for resources, as opposed to those caused by faulty synchronization of concurrent tasks, since the latter are more complex and less well understood.

REFERENCES

1971. Coffman, E. G., M. J. Elphick, and A. Shoshani. "System Deadlocks," *Computing Surveys*, Vol. 3, No. 2 (June).
1972. Holt, R. C. "Some Deadlock Properties of Computer Systems," *Computing Surveys*, Vol. 4, No. 3 (September).

A. H. WERKHEISER

DEBUGGING

For articles on related subjects see **COMPUTER**, **USING A**; **DUMP**; **ERRORS**; **FLOW-CHART**; **PATCH**; **PROCEDURE-ORIENTED LANGUAGES**, Programming; **PROGRAM**; and **TRACE**.

For article on related term see **LOOP**.

When a program is first ready to be tried out on a computer, the probability is very high that it contains one or more mistakes, introduced at any of several places along the line of preparation. Debugging is the process by which a programmer finds and then corrects the errors in a program. These mistakes may take various forms, such as merely a keypunching mistake; a grammatical error due either to an oversight or to misuse of the syntax of the language; a logical error resulting either in an inability to proceed with the calculation or in proceeding with the wrong calculation; or a choice of method that turns out not to be valid for the particular circumstances. Anticipating this to be the case, the programmer proceeds to establish, with the

aid of the computer, the answers to three questions about the program:

1. Is it a valid program? Do the statements, individually and together, follow the syntactical rules of the language so that they are capable of being translated into a machine language program?
2. Is it a *working* program? Can the computer execute the program so translated?
3. Is it a correct program? Will it accomplish the desired goal? Are the answers produced correct?

Question 1: Is It a Valid Program?

All compilers determine this. Merely by trying to translate the given program, of necessity they uncover all instances of bad grammar, whether it is a misplaced comma, improper usage, or missing information. Of course, when the programmer makes a mistake resulting in a statement that is syntactically correct but different from what he intended, the compiler can do nothing; but when an error in syntax is encountered, it is almost always caught, flagged, and a diagnostic message put out.

Compilers vary greatly in the degree of help they give in such situations. In the early days, when memories were limited, merely an abbreviated error code was often printed, requiring the user to look up in a table or listing to find out the kind of error involved. Now, most compilers print **textlike** messages, although that does not guarantee their clarity.

For example, omission of an operation symbol is easy to catch, and easy to point out, as in the example :

```
D = B*B - 4.AC
****          $
IMPLIED MULTIPLICATION NOT ALLOWED--USE *
```

However, compiler writers for commercial companies seem rarely to be concerned with the confusion or blankness in the mind of the novice programmer. It is certain that a beginner who forgets to **DI-MENSION** an array (but who has never heard of a statement function) will not find the message

```
DO 1 I = 1, 3
1 A(I) = I
FATAL ERROR:
THIS DEFINE PROCEDURE IS DEFINED AFTER AN
EXECUTABLE STATEMENT
```

very clarifying, and will have to go elsewhere for (human) help. In this respect, the writers of many university-developed compilers, like Watfiv and Fortgo, have done far better at anticipating the state of

twareness of the programmer, and have tried to give him more helpful information. For the preceding mistake, for example, Forgo says:

```
DO 1 I = 1, 3
1 A(I) = I
$
VARIABLE IS NOT DIMENSIONED
```

Sometimes, of course, even the best compiler is reduced to frustration and has to say:

UNDECODEABLE STATEMENT

Nevertheless, most compilers of today's generation, by and large, do an adequate job (for the experienced programmer, at least) of helping to find mistakes in this first category.

Question 2: Is It a Working Program? Assuming that there is a unique interpretation that can be found for every statement, control can be passed over to the resulting object program, and execution can be attempted. If the computer encounters an impasse at execution time—a situation where insufficient information exists, or contradictory instructions are issued—it is usually due to programmer inconsistencies: failing to make sure every quantity is defined (has a value assigned to it) before asking the computer to use it; exceeding the range or the bounds he himself has defined for values, subscripts, or arrays; contradicting himself by supplying the wrong type of data or by calling for a mathematical process that is undefined for the particular values being used, like division by zero.

The response given by different processing systems varies widely here. The error may be one of three types:

A. *Fatal*—execution is terminated.

EXAMPLE 1. A common mistake of beginning programmers (and all too frequently of experienced ones as well) is neglecting to initialize a variable being used iteratively, such as when accumulating a sum. In such a case, the first time the computer is asked to use the quantity, it is told to take the contents of a storage location to which this program has not yet assigned a value. Forgo detects this and stops; a warning message is put out, identifying the situation, the variable, and the location in the source program where the usage occurred.

```
DIMENSION A(3)
DO 1 J = 1, 3
A(J) = J
```

```
1 SUM = SUM + A(J)
PRINT, SUM
STOP
END
```

\$LINKGO

```
ERROR IN *MAIN* AT STMT. NO. 00001 + 00
LINES :
SUM      UNDEFINED VARIABLE
```

ABORT

EXAMPLE 2. When dealing with arrays, it is all too common for a programmer to fail to keep track of where his subscripts are ranging and, in particular, to let them stray beyond the bounds he himself has declared for them. Again, a good diagnostic processor will catch this and tell the programmer about it.

```
DIMENSION N(3)
DO 1 J = 1, 3
1 N(J) = J
DO 2 K = 1, 4
2 PRINT, K, N(K)
STOP
END
```

\$LINKGO

```
1      1
2      2
3      3
```

```
ERROR IN *MAIN* AT STMT. NO. 00002 + 00 LINES :
N      ERROR IN SUBSCRIPTING
```

ABORT

B. *Nonfatal*—an interpretation is invented, a warning issued, and processing continued.

EXAMPLE 3. The square root of a negative quantity is undefined in real mathematics. What should the processor do when the argument of a **SQRT** function turns out to be negative? As long as a clear warning is given, almost anything can be used. The important thing is that even a computation with incorrect results may give useful information during debugging; therefore, it is better to do something and continue the computation than just to stop.

```
00102 x = -4.
00103 Y = SQRT(X)
00104 PRINT, X, Y
```

DEBUGGING

```
*****
ERROR DETECTED IN SQRT ROUTINE
NEGATIVE ARGUMENT , ARG1 = -.40000000+01
SQRT CALLED AT SEQUENCE NUMBER 000103
OF MAIN PROGRAM
*****

-4.0000 0.0
```

EXAMPLE 4. Sometimes the computer is asked to use the value of an undefined variable only in an output statement. In such a case, some processors will print out a unique and identifiable string (e.g., in Watfiv, a number known as “standardized garbage”; in Fortgo a string of `□□□□s`), and proceed. The justification for this is that such output use of a wrong value may still allow useful calculations that do not involve that value.

```
DIMENSION N(4)
DO 1 J = 1, 3
1 N(J) = J
DO 2 K = 1, 4
2 PRINT, K, N(K)
STOP
END
```

```
1          1
2          2
3          3
4 □□□□□□□□□□
```

C. Ignored-an interpretation is invented, no warning issued, and processing continued!

EXAMPLE 5. In some Fortran language adaptations, such as Fortran G on the IBM 360, the processor takes whatever value was last stored in the specified location by the preceding program, and goes blithely on.

```
DIMENSION A(3)
DO 1 J = 1, 3
A(J) = J
1 SUM = SUM + A(J)
PRINT, SUM
STOP
END
```

550.9134

EXAMPLE 6. In Fortran V on the Univac 1108, the processor at least clears the memory being used to 0's before starting each program, and, if no other value has been defined, zero will be used.

```
DIMENSION N(4)
DO 1 J = 1, 3
1 N(J) = J
DO 2 K = 1, 4
L = N(K)
2 PRINT, K, N(K)
STOP
END
```

```
1
2          2
3          3
4          0
```

Some programmers may, having learned this, actually count on it and use this feature intentionally. In neither case is the programmer given any warning that the processor has made an assumption and proceeded in spite of incomplete instructions. This, unfortunately, is the case for the great majority of commercially available compilers.

Furthermore, sometimes the invented interpretation will lead to destruction of parts of the program or data, resulting in a later error message that seems to have nothing to do with the original error.

Question 3: Is It a Correct Program?

If answers are produced by the program and no diagnostic message appears that would invite suspicion, the programmer must now face the question of whether the answers are correct. Here, unfortunately, the computer can do little to help, at least on its own initiative. That is, very little can be built into a processor to operate automatically in this situation. The judgment of the programmer must be used to decide whether the results are acceptable, incorrect, or of uncertain validity. There are, however, several additional techniques and special software features that can be made available to the programmer to assist him in this process of getting the bugs out, or “debugging,” as it is commonly called.

If, as often happens, the program does not appear to be working as the programmer thinks it should-if it never produces output from a certain section, or appears to be repeating certain sections too many times (called “looping”)-a simple procedure is to insert some extra output statements at judicious spots in his program, so that certain intermediate values are printed out for inspection. These may be the results of certain steps, or merely counters or index values. When the troubles are fixed, the extra statements can then be removed, or

\$LOADGO FORGO

DIMENSION DATA (I 0)

***TRACE**

CALL TRACE (3)

READ, N, (DATA(1), I = 1, N)

CALL TRACE(0)

DO 1 I = 1, N

IF (I.GE. (N - 2)) CALL TRACE (3)

SUM = 0.

SUM = SUM + DATA(1)

1 CONTINUE

AVE = SUM/N

PRINT, SUM, N, AVE

STOP

END

\$LINKGO

```

*MAIN* 00000 + 03      7 1.23 4.56 -7.32 45.32 -9.2 6.89 -2.50
      00000 + 07 0.0000 SUM      00000 + 08 -9.2000E + 00 SUM      00000 + 05      6      |
      00000 + 06 T
      00000 + 07 0.0000 SUM      00000 + 08 6.8900E + 00 SUM      00000 + 05      7      |
      00000 + 06 T
      00000 + 07 0.0000 SUM      00000 + 08 2.5000E + 00 SUM      00001 + 01 -3.5714E - 01 AVE

2.5000      7      -3.5714E - 01
  
```

Fig. 1. Output of a trace program.

merely jumped around.

To simplify this process, many processors offer more or less elaborate capabilities to "trace" a program. Trace programs will produce, usually on the output printer, dynamic information about the progress of the program, such as the flow of control from one part to another, the new value of a variable every time it is changed, index and counter values on each pass through an iteration, and the value of an expression in an **IF** statement to help see why the program jumped or did not jump. Indiscriminately used, tracing can be extremely expensive, both in the slower execution necessary while producing tracing information, and in the huge volume of output that may be spewed forth. A well-designed general trace package, however, permits the programmer not only to specify at compile time in which parts of the program he would like tracing instructions included, and which variables and arrays he wants traced, but at execution time he can turn the tracing on and off at will.

An example of the output of such a trace program is shown in Fig. 1, produced by the Datacraft version of Forgo developed at the Uni-

versity of Wisconsin. The ***TRACE** card is a compile-time instruction, and causes trace instructions to be compiled with the rest of the object program. It could have been inserted part way through the program if interest had been centered farther on; **CALL TRACE (N)** is an execution-time instruction, which turns on the production of trace output, N items per line, if $N > 0$, and turns off the production of output when $N = 0$. For each assignment statement traced, three items of information are printed: (1) the location of the statement, identified in terms of statement numbers and lines counted beyond the last statement number (e.g., 00001 + 01 is the first statement following statement number 1; statement 00000 + 01 is the first executable statement in a program, if unnumbered); (2) the value of the variable named on the left, after execution of the statement; and (3) the name of the variable or array element. If an **IF** statement is traced, the value of the expression is printed, without a name: a numerical value for an arithmetic **IF**, a T or F for a logical **IF**. For each **READ** statement traced, the location is given, followed by a copy of the data read.

Note that listing of output is suppressed by the

DEBUGGING

\$LOADGO FORGO

```
DUMP
DO 1 N = 1,100
  READ, X, Y
  Z = DIFF(X, Y)
  IF(Z.GT..05)PRINT, X, Y, Z
1CONTINUE
STOP
END

FUNCTION DIFF(R1, R2)
DUMP
DIFF = ABS((R1 - R2)/R1)
RETURN
END
```

\$LINKGO

4.0000	1.0000	7.5000E-01
6.0000	9.0000	5.0000E-01
7.0000	-3.0000	1.4286
4.0000	0.0000	1.0000

ERROR IN DIFF AT STMT. NO. 00000+ 03 LINES : DIVISION BY ZERO
CALLED IN *MAIN* AT STMT. NO. 00000+ 04 LINES:

MAIN

x = 0.0000 , Y = 5.0000E+00, N = 5, Z = 10000E+00

DIFF

R1 = 0.0000 , DIFF = 1.0000E+00, R2 = 5.0000E+00,

ABORT

Fig. 2. Output of a Forgo DUMP program.

programmer until the loop is nearing satisfaction, since values near the end are often the critical values. In this case, however, inspection of the trace output quickly reveals (if it had not been apparent before) that the initialization statement has been erroneously included in the loop, and the sum has been reset to zero on each iteration.

Sometimes, even with tracing available, it is difficult to get a clue as to where something is going wrong in a program. It may terminate abruptly with only a cryptic **ABEND**, or “abnormal termination” message, indicating that “something went wrong somewhere.” In such cases the instruction to produce a **DUMP**, or “post-mortem,” showing the status of both program and data at the time the program terminated, may yield valuable information. Systems that produce this information only in terms of core

locations or absolute addresses necessitate the use of a “symbol table map” that enables the programmer to determine, at the expense of only a few hours and a finite fraction of his remaining eyesight, what value belonged to what variable, and where the program might have been when it blew up.

Some maps actually give the values in binary or octal code, putting the even more tedious burden of transliteration on the program user. Better designed systems, however, will list selectively the values of only those variables or arrays specified, and will furthermore identify each with its name in source-language terms. Fig. 2 (again taken from Forgo) shows an example of such a **DUMP**. In this case, the trouble can easily be determined to be an attempt to divide by zero when the fifth set of data is processed. Although in this case it would have been easy to

DEBUGGING

1:	WRITE(6,10)	SYMBOL	REFERENCED AT LINES (MINUS MEANS SYMBOL DEFINED,
2:	1 READ, A, B, C	NAME	EXCLUDING SUBPROGRAM CALLS AND EQUIVALENCE
3:	IF(A .NE. 0) GO TO 2	SORT	24 33
4:	IF(B .EQ. 0) GO TO 7	S	-24 25 26 -33 35
5: c		RR	-34 37
6: c	EQUATION IS LINEAR	ROOT2	-26 28
7: c		ROOT1	-25 28
8:	ROOT = -C/B	ROOT	-8 9
9:	WRITE(6,11) A, B, C, ROOT	RI	-35 37
10:	GOTO1	D	-19 20 24 33
11: c		C	-2 8 9 14 19 27 36
12: c	ERROR	B	-2 4 8 9 14 19 25 26 27 34
13: c			36
14:	7 WRITE(6,16) A, B, C	A	-2 3 9 14 19 25 26 27 34 35
15:	GO TO 1		36
16: c			
17: c	TEST DISCRIMINANT		
18: c			
19:	2 D = B*B - 4*A*C	STATEMENT	DEFINED
20:	IF(D .LT. 0.) GO TO 4	NUMBER	AT LINE REFERENCED AT LINES
21: c		1	2 10 15 29 38
22: c	EQUATION HAS TWO REAL ROOTS	2	19 3
23: c		4	33 20
24:	S = SQRT(D)	7	14 4
25:	ROOT1 = (-B + S)/(2*A)	10	39
26:	ROOT2 = (-B - S)/(2*A)	11	40 9
27:	WRITE(6,12) A, B, C	12	42 27
28:	WRITE(6,13) ROOT1, ROOT2	13	43 28
29:	GOTO1	14	44 36
30: c		15	45 37
31: c	EQUATION HAS TWO COMPLEX ROOTS	16	47 14
32: c			
33:	4 S = SQRT(-D)		
34:	RR = -B/(2*A)		
35:	RI = -S/(2*A)		
36:	WRITE(6,14) A, B, C		
37:	WRITE(6,15) RR, RI		
38:	GOTO1		
39:	10 FORMAT(6X,'A',8X,'B',8X,'C'/1X,3('-----'))		
40:	11 FORMAT(3F9.2,' IS A LINEAR EQUATION'/		
	* 36X,'ROOT = ',1PE11.4/)		
42:	12 FORMAT(3F9.2,' HAS TWO REAL ROOTS')		
43:	13 FORMAT(38X,'R1 = ',1PE11.4/38X,'R2 = ',1PE11.4/)		
44:	14 FORMAT(3F9.2,' HAS TWO COMPLEX ROOTS')		
45:	15 FORMAT(36X,'REAL = ',1PE11.4/		
	* 31X,'IMAGINARY = ',1PE11.4/)		
47:	16 FORMAT(3F9.2,' NO EQUATION')		
48:	END		

Fig. 3. Cross-reference output.

DEBUGGING

detect by an inspection of the data, in other situations the trouble might have arisen from numbers that were the results of intermediate calculations, and might be much more difficult to track down.

One more type of program aid is sometimes of considerable value in debugging, both after trouble develops, and even more to help keep troubles from getting started. This is a "cross-referencing" program that will produce indexed lists of both the variable names and the statement numbers of a source program, which may be of great value both in tracing the flow through the program and in determining where each variable is used, including which statements have the capability of defining a new value for it. Fig. 3 shows the cross-reference output produced for a simple quadratic equation solving program.

Such a cross-referencing program is of particular value when modifications need to be made to an existing program, either a working program or one still being debugged. It allows a backward trace of the flow path by showing the possible paths that might have been followed to get to a particular point; it tells precisely all the places where a particular variable is used, and which of those are reference usages and which are defining usages. Methodical reference to a cross-referencing program can greatly decrease the incidence of second-order bugs—those introduced in the process of trying to fix other bugs.

Besides the use of special processors like tracing and dump programs, programmers often employ all or some of the following techniques, developed out of painful experience, to facilitate debugging their programs:

1. Since most modern compilers will produce a source-language listing of the program as it is being compiled, but not of the data being supplied with it, it is frequently good practice to include an output statement immediately following every input statement, which will print out the values just read. The name "echo checking" or "echo printing" is sometimes applied to this.

2. It is worthwhile for debugging purposes, as well as for the usefulness of the eventual results, to be sure each output value is properly identified by appropriate column headings, captions, or labels, including all independent variable values, count numbers, and other such information. Even though it may be possible to identify an unlabeled output value with a little consideration and comparison, careful anticipation of the appearance of the output page and of the ease and quickness with which

meanings can be associated with numbers can greatly speed up comprehension.

3. Liberal *use* of comment cards in the source program is necessary if anyone else is to make use of the program. It is also extremely helpful to the author himself in the process of debugging when he tries to answer such questions as, "What does this part of the program do?"; "Where is the section that does...?"; "What does XYGRAB represent?"; or even, "How did I get here?" Flowcharts carefully correlated with the program in terms of segments, blocks, statement numbers, etc., can also be invaluable in helping to answer such questions.

4. A little extra time spent in careful organization and layout of the source program itself can often save much valuable time in the debugging process. Indentation of subprograms, clear identification of the end of such program segments, and vertical alignment of alternative conditions in branching operations are a few suggestions (see McCracken and Weinberg, 1972).

Such techniques might be summed up in a fifth one:

5. The more care is taken at program construction time, the less hair pigmentation is lost in debugging.

This article has focused on debugging techniques for **Fortran** programs on some well-known university systems. Although such systems tend to have better than average debugging facilities because they are intended for student use, all higher-level language systems have some debugging facilities of the type described here. Assembler and **macro**-assembler systems also all have debugging facilities whose details will generally differ from those presented here, but whose spirit—in terms of getting answers to the three questions posed at the beginning of this article—is quite similar to that discussed here.

Some aspects of debugging are beyond the scope of this article. One of these is the increasing use of interactive time-sharing systems for debugging. Although many of the techniques described here are also used in interactive debugging, a number of interactive debugging techniques have been developed to take advantage of the on-line dialog which is possible between the programmer and the computer. Another subject beyond the scope of this article concerns the debugging of large software systems that present problems to the programmer quite different in scope than the debugging of most higher-level language programs (Rustin, 1971).

REFERENCES

1971. Rustin, Randall (Ed.). *Debugging Techniques in Large Systems*. Englewood Cliffs, N.J.: Prentice-Hall.
1972. McCracken, Daniel D., and Gerald Weinberg. "How to Write a Readable Fortran Program," *Datamation*, Vol. 18, No. 10 (October), pp. 73-77.
1973. Hetzel, W. C. (Ed.). *Program Test Methods*. Englewood Cliffs, N.J.: Prentice-Hall+

C. H. DAVIDSON

How the notion of decidability of a predicate can be made mathematically rigorous and how the decidability of predicates can be discussed in terms of the computability of functions are explained in the article Algorithms, Theory of

REFERENCE

1969. Hopcroft J. E., and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.

G. T. HERMAN

DECIDABILITY

For articles on related subjects see **ALGORITHMS**, **THEORY OF**; and **COMPUTABILITY**.
For article on related term see **ALGORITHM**.

Decidability is a property of predicates. A *predicate P* with domain *D* is a property of the elements of *D*, which each particular element of *D* either has or does not have. If *x* in *D* has the property *P*, we say that "*P(x)* is true"; otherwise, we say that "*P(x)* is false." The predicate *P* is said to be *decidable* if there exists an algorithm which, for any given *x* in *D*, provides us with a definite answer to the question whether or not *P(x)* is true.

For example, consider the predicate *P* whose domain *D* is the set of integers greater than 1, and which is defined as:

P(i) if and only if *i* is a prime number.

This predicate is decidable. An algorithm is described by Hopcroft and Ullman (1969).

The definition given above is lacking rigor for the following reasons:

1. It is not explained in what form the argument *x* in *D* is "given." In particular, this part of the definition makes sense only if elements of *D* are in some way finitely describable. Thus, the notion of decidability, as described above, makes no sense for a predicate *P* whose domain is the set of real numbers.

2. The notion of an algorithm is not precise.

3. It is not explained in what sense the algorithm provides us with an answer.

DECISION TABLES

PRINCIPLES

For articles on related subjects see **DECISION TABLES** : Languages ;and **FLOWCHART** .

Decision tables are a tabular method of describing or specifying the various actions associated with combinations of conditions. The method is tabular in that it uses a special form of table to present the associations. The actions specified are transformations to be done to data or materials, usually by computers or people. The conditions are data variables that describe the characteristics of the environment and the events that happen in the environment. The relationship among the conditions specified in a decision table is usually the logical **AND** relationship. The history of the origin and development of decision tables is summarized in Chapin (1967).

EXAMPLE. A simple example of a decision table is given in Fig. 1. This describes a procedure for ordering low-usage products under several conditions. By policy, a target inventory level has been set at 20 units of stock for items covered by this decision table. Consider, as an example of the procedure, the third column from the right in the decision table: If the weekly usage is low (less than 8 units) and the amount on order is not greater than 30 units, then a regular order should be placed if the stock on hand amounts to less than 20 units.

Terminology. As indicated in Fig. 2, decision tables are commonly regarded as consisting of four overlapping major parts. Each of these parts is a

DECISION TABLES

On hand < 20	Y	Y	Y	Y	Y	Y	N	E
Weekly usage	> 15	> 15	8-15	8-15	8-15	< 8	-	S
Local vendor available	-	-	N	N	Y	-	-	L
On order > 30	N	Y	N	Y	N	N	Y	E
Rush order	X		X					
Regular order		X		X	X	X		
Cancel order							X	
No action								X

Fig. 1. Example of a decision table.

rectangle within the overall rectangle of the decision table. The proportioning of the four-component rectangles varies from situation to situation.

STUB	CONDITION
ACTION	RULES ENTRIES

Fig. 2. Parts of a decision table.

The upper portion of a decision table is known as the "condition" portion.* The lower portion of a decision table is known as the "action" portion. The left-hand portion of a decision table is known as the "stub." The right-hand portion of a decision table is known as the "entry" portion. Each column in it is known as a "decision rule." Because of the overlap, the upper left-hand portion of a decision table is known as the "condition stub." The upper right-hand portion is known as the "condition entries" in the decision rules. The lower left-hand portion is known as the "action stub." The lower right-hand portion of a decision table is known as the "action entries" in the decision rules.

Decision rules are any of the columns in the rules or in the entry portion of the decision table. The rules in decision tables are meant to be read both horizontally and vertically. Thus, read **verti-**

cally, each decision rule cites some combination of conditions and the associated actions to be taken when that combination of conditions is true (exists or is satisfied). Read horizontally, the rules list the alternative values of conditions and the presence or absence of the actions to be taken. The stub, read **vertically**, lists all conditions and actions. Taken together, the decision rules are exhaustive in that they must cover every possible combination of conditions. No such requirement is imposed on the actions, however. Conditions may be of three types: limited entry, extended entry, or mixed entry.

In *limited-entry decision tables*, the condition stub specifies exactly what the condition is, or what the value of the variable is. An example is, "Age is less than 18." Therefore, the condition portion of each decision rule may need only to identify if yes (Y), that condition is met; no (N), that condition is not met; or "don't care" (-), whether or not the condition is met. In the latter case, the rule is insensitive or indifferent to the particular values of that condition.

In *extended-entry decision tables*, the condition stub cites the identification of the condition, but not the particular values. Particular values are entered into the condition portion of the decision rules directly. An example is "Age" in the stub, and "< 18," "18," and "> 18" in the rules.

In *mixed-entry*, the action portion of the decision rules may be either unsequenced or sequenced. Unsequenced actions, the most common, are identified by any mark (such as X) in the rule, with the actions to be performed in the order in which they are listed from top to bottom in the action stub. If needed, actions may be cited more than once in the stub, to get them into the desired sequence for a

decision rule. Actions not to be performed are left unmarked in the rule. For sequenced actions, the action entry is a sequence number instead of a mark, and the action stub may list the actions in any sequence.

Reading and Creating Decision Tables. The person who creates a decision table must give attention to completeness, accuracy, redundancy, inconsistency, endless loops, and size. For assistance on the first four matters, formal guides and check procedures are available [for examples, see London (1972) and Pollack et al. (1971)]. To avoid all endless loops, the person must require for every decision rule that at least one of the marked actions must change the value of at least one of the conditions cited in the condition stub.

In order to save space in the decision table, and to keep the decision table down to a workable size, rule consolidation is normally practiced as often as possible. This is usually accomplished in two ways. First, when the actions to be taken for different combinations of conditions are identical, and the patterns of conditions can be combined through the use of "don't care" condition entries, then one decision rule can replace two or more decision rules. Second, an "else rule" can be specified for all possible combinations of conditions not explicitly provided for in the other decision rules. Commonly, the else rule is the rightmost rule in a decision table (see, for example, Fig. 1). Large decision tables can be split into a connected group of much smaller decision tables by using decision table parsing techniques.

The user of decision tables commonly reads the decision rules individually from top to bottom, referring to the left to the stub as needed. Except for the "else" rule, the left-to-right sequence of the decision rules in the decision table is of no significance, but is commonly put into a logical order based upon the pattern of changes in the conditions. Having the most commonly used rules at the left is an aid to the user, but having them in an orderly progression of condition values is an aid to the creator of the decision table.

To use a decision table, the user first reads the condition stub and, for the situation at hand, notes the input values for each condition. Then he matches these values of the conditions against the condition portion of the decision rules, one rule at a time, from left to right. If the conditions do not match, the user rejects the decision rule and goes to the next decision rule to the right. If the table was correctly created, one and only one decision rule must fit the input

values of the conditions, be it only the "else" rule. When the user finds the rule that does match, he accepts the rule and goes to its action portion to find out what actions are to be performed and in which sequence. When these actions are complete, the user applies the decision table afresh with a new set of input values for each condition.

Use. Decision tables have enjoyed their widest use in representing logically complex data-handling situations. These are situations where the actions to be taken depend upon the values of a large number of variables, taken in combination. Examples of such situations are commonly encountered in administrative and control applications of computers. Major users include insurance companies and other financial organizations, and manufacturing companies (McDaniel, 1970). Decision tables are rarely used in scientific or research organizations when the situations can be clearly described in mathematical terms. Having or using a computer is not a prerequisite to using decision tables.

Decision tables find use in many phases of computer work, including system analysis, system design, programming, debugging, and documentation. In systems analysis, decision tables help analysts in identifying the significant control variables for the operation being studied. In systems design, decision tables help link the desired action to the control variables. In programming, decision tables can be used as a programming language. In debugging, decision tables can help reduce the time to locate bugs because they force a sharp distinction between control logic in a program and the actual production of the output data. In documentation, decision tables can concisely summarize the system or program in written form.

Advantages. Decision tables can serve as a compact means of describing or specifying operations. How compact they are depends on the number of conditions being included, and on the number of possible different actions that need to be taken. In general, the compactness of decision tables decreases about in proportion to the sum of the number of variables included and the number of possible actions.

Because of its compactness, the decision table provides a convenient way of tersely stating logically complex processing. The practical size for a single decision table is approximately what can be put on one page of paper. Larger decision tables can be parsed and linked together. The procedures for creating decision tables provide rules for checking for

DECISION TABLES

four types of possible errors: completeness, size, redundancy, and inconsistency (London, 1972; Pollack et al., 1971). These offer valuable aids in systems design and programming, but people must go through the laborious process of doing most of the checking work.

Decision tables can be used to summarize much information in documentation, but in this case are sometimes regarded as being too concise. In programming, their precision and conciseness are major advantages, when supported with additional documentation.

Disadvantages. Decision tables have no theoretical size limit, but there are real practical limits imposed by people. Large decision tables become incomprehensible, and can be neither checked nor used well by people. Fortunately, the size can usually be reduced by rule consolidation and by parsing.

Decision tables do not reduce the human labor of thinking or discovery. Human beings still must do the work of defining, specifying, and following to its logical consequences each chain of conditions and actions. Decision tables take away from people none of this arduous work, but they can be used to pinpoint where that work can be best concentrated.

Decision tables ignore the delicate interleaving of logic and action that seems so natural when people think about conditions and actions to be taken. Decision tables force the human user to consider conditions separately from actions.

The advantages of decision tables are drawing an increasing number of supporters, but their disadvantages are sufficiently major to keep this group of users fairly small. Typically, the experience of the first-time user is that he must increase the time and effort he puts in, in order to prepare the decision table. This additional investment may pay off in less debugging and in more efficient operations, as is usually the case for the experienced user of decision tables, but the additional investment by the novice user is difficult to justify.

REFERENCES

1967. Chapin, Ned. "An Introduction to Decision Tables," *DPMA Quarter*, Vol. 3, No. 3 (April), pp. 2-23.
1970. McDaniel, Herman (Ed.). *Applications of Decision Tables*, Philadelphia: Auerbach Publishers.
1971. Pollack, Solomon L., Harry T. Hicks, and William J. Harrison. *Decision Tables: Theory and Practice*. New York: John Wiley.

1972. London, Keith R. *Decision Tables*. Philadelphia: Auerbach Publishers.

N. CHAPIN

LANGUAGES

For articles on related subjects see **DECISION TABLES**; Principles; **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.

For articles on related terms see **DIAGNOSTICS**; and **SOURCE PROGRAM**.

A decision-table language is a higher-level programming language, which in major part has the appearance of, and also serves as, a decision table. A decision-table language is a part of a decision-table programming system, with the decision-table language supported by a translator or processor (or a series of them) to produce executable machine language code.

Decision-table languages differ from other higher-level languages in several respects. First, the syntax, while commonly very close to Cobol or Fortran, goes beyond those languages to allow source language statements, which taken together look like a decision table. This permits the programmer to write a program in decision-table form in part, as illustrated in Fig. 1.

When the analysis and design work have resulted in a decision table, using a decision-table language avoids having the programmer translate the decision table. This greatly speeds up programming and reduces logic bugs. When the analysis and design work have not resulted in a decision table, using a decision-table language for the programming helps provide a check on the completeness and consistency of the design.

Second, the supporting translators or processors often first generate or output Cobol or Fortran source language. In these cases, the decision-table language effectively becomes a preprocessor, an added "front end" to, or alternative way of producing, Cobol or Fortran programs. Third, decision-table languages show great diversity among themselves, enough so that what is acceptable syntax in one is commonly unacceptable in most of the others. Fourth, computer vendors have commonly not provided translators or processors to support decision-table languages. The result has been a

profusion of decision-table programming systems from major users and from software houses.

Background. Because of the third and fourth points noted above, any brief discussion of this subject must of necessity fail to give a full picture of the convenient and powerful features available in some decision-table languages. The variety is just too much to cover both adequately and briefly. For this reason, a summary discussion such as this one must concentrate on only a selection of features and on the varieties of the decision-table languages likely to be most commonly encountered or likely to be of most significance to most users of higher-level languages. A look at the history of decision-table languages helps clarify this situation.

The first decision-table language was **Tabsol**, produced in 1961 by the Computer Department of General Electric Company (Chapin, 1967). Its translator went directly to machine language, but by comparison with later developments, the syntax was very limited. In 1962 and 1963, a CODASYL committee proposed a decision-table language extension or addition to Cobol. This effort, **Detab**, appeared first in an unsupported experimental version **Detab-X** (CODASYL Systems Group, 1962). This was later (1963-1965) refined to **Detab/65** and supported by a decision-table-to-Cobol translator. For some years, the U.S. Navy has distributed an improved version of **Detab** called **DTT**, and in 1972 offered a new translator to Cobol, itself written in Cobol.

Concurrently with these developments, a host of other decision-table languages appeared, as well as variations of **Detab**. Among the later were offerings by IBM, Univac, CDC, Honeywell, and ICL. Among the former, from computer vendors, were offerings by General Electric (**Logtab**), and IBM (**DLT**, Decision Logic Translator). From users and software houses came many offerings. Among them were **Detap** from Information Management Inc., **Tabtran** from Westinghouse Tele-Computer Systems Corp., and **SMP** from Trilog Associates Inc., and **Tap** from Hoskyns Systems Research in Europe. In 1973, the total number of active decision-table languages exceeded two dozen, but only those listed above were popular. A 1969 summary of the major offerings to that year listed 32 decision-table languages (McDaniel, 1970).

Features. A comparison of a selection of decision-table languages, based on their major features, helps indicate their variety and range of usefulness. The languages stressed in this compar-

ison are **DTT**, **Detab**, **SMP**, and **Tabtran**. Where other decision-table languages significantly differ on the points of comparison, note is taken in most cases.

Source Language. The programmer writes a decision table on the coding sheet, following some format rules. In some languages, such as **DTT**, he must provide a header line to describe the size and form of the decision table. In most, he provides no header line, but either must follow a prescribed format (as in **SMP**, for example) or must follow punctuation rules to enable the translator to identify the parts of the decision table (as in **Detab**, for example). In all, the programmer must take care in selecting the wording in the stub.

Target Language. Nearly all modern translators produce as their output a source language translation of the table input. Most produce Cobol source code, although some, such as **Tabtran**, may optionally produce Fortran source code. A few languages produce low-level symbolic or assembly language as the target language.

Output from Translator. The translator produces a network of comparisons with conditional and unconditional transfers of control to replace the decision rules. The translator usually does not replace the stub at all, but uses the programmer-provided wording directly in the comparisons and as the actions. This means that symbolic names must be fully defined by the programmer or specified in the syntax, be they control entry points, names of items of data, or relations (such as "greater than"). To provide control entry points in the comparison network, the translator usually provides catenated synthetic names. The action-stub names usually become control entry points for exits from the comparison network.

Entry Form. All decision-table languages accept limited-entry decision tables, and a few such as **Detap** accept extended and a mixture of limited and extended entry forms.

Number of Rules Permitted in a Table. The number of decision rules permitted as the maximum has been as follows: **DTT**, 25; **Detab**, 50; **SMP**, 20; and **Tabtran**, 40.

Else Rule. **DTT** requires the programmer to provide an "else" rule, even though it be only a dummy. For the others, the "else" rule is optional. **Detab** generates an "else" rule if one is needed but is not provided by the programmer.

Number of Conditions. The number of conditions permitted as the maximum has been as follows: **DTT**, 25; **Detab**, 50; **SMP**, more than 1000; and **Tabtran**, 100.

Number of Actions. The number of actions per-

DECISION TABLES

[illegible]

DECLARATIVE STATEMENT

```
00104 000770  DISPLAY 'ELSE RULE NONE SPECIFIED-TBL = 01 O-DECIDE'.
00105 000772  STOP RUN.
```

FIXPOP
FIXPOP

Fig. 1. Decision-table in the Detap programming language. This is a replica of a decision table re-formatted into Cobol. Note that Paragraph (OP) is now part of the Cobol program and is immediately followed (FIXPOP) by the Cobol source code generated by Detap. (Courtesy of Information Management, Inc.)

mitted as the maximum has been as follows: DTT, 25; Detab, 50; SMP, more than 1000; and Tabtran, 150 minus the number of conditions.

Sequencing Of Actions. Most provide for unsequenced actions. A few, such as SMP, provide only for sequenced actions; and a few, such as Detab, provide for both sequenced and unsequenced actions.

Diagnostics. The variety of diagnostics available from the translators is extensive, covering not only syntax matters but also reports on logical errors in the table. Almost all apply tests for rule redundancy and for rule inconsistency. Tests for rule completeness depend in part on the handling of the "else" rule, as do some tests for accuracy. Hence, some languages, such as DTT and Tabtran, provide no diagnostics in these areas, whereas others, such as Detab and SMP, do.

Size Reduction. Most of the decision-table translators, as a byproduct of testing for rule redundancy and inconsistency, also attempt to consolidate rules by the use of the "don't care" entry. A minority, including DTT (at least with its prior to 1972 translator), generated equivalent code for rules that could be consolidated. None of the decision-table languages provides for parsing, and all rely on the host language (usually Cobol or Fortran) for the handling of any linkage between tables.

Other Aids to Users. Some of the translators are limited to just the handling of the translation, such as DTT. Others, such as Detab, can tie into a larger software package. A few, such as SMP, make the handling of decision tables the keystone to a system of program creation and maintenance, and provide such features as libraries of program versions and cross-references to data names and control entry points.

Configuration. To run the translators, the usual practice is to make the minimum computer configuration needed be the same as that required for the host language, usually Cobol or Fortran. Most are available for running under any of the common operating systems that support the host language.

Optimization. The effectiveness of the optimization of the object code possible from the translators is moot. In part, it depends on the compiler for

the host language. In part, it depends on the algorithms used by the decision-table translator, the effectiveness of which have been questioned. In part, it depends upon the factor chosen for optimization, since the factors are antagonistic (speed of execution versus storage space needed, for example).

Price and Availability. Decision-table languages and translators are readily available for immediate delivery, and many have been extensively field tested. Most require a periodic monthly license payment, usually between \$200 and \$500 per month. Some are available for a period, usually three years, for a one-time license payment of from \$1,000 to near \$5,000. A few are available only on a purchase basis, at prices from nearly \$100,000 to almost free, such as DTT.

REFERENCES

- 1962. CODASYL Systems Group. *DETAB-X*. Santa Monica, Calif: CODASYL Systems Group.
- 1967. Chapin, Ned. "An Introduction to Decision Tables," *DPMA Quarterly*, Vol. 3, No. 3 (April), pp. 2-23.
- 1970. McDaniel, Herman. *Decision Table Software*. Philadelphia: Auerbach Publishers.

N. CHAPIN

DECLARATIVE STATEMENT

For articles on related subjects see **EXECUTABLE STATEMENT**; **PROCEDURE-ORIENTED LANGUAGES**; **PROGRAMMING LANGUAGES**; and **STATEMENTS**.

A declarative statement is one in a higher-level programming language that provides descriptive information (contrasted with an imperative statement that specifies explicit processing operations).

Besides specifying the actual computations, decision rules, and input/output operations involved in

DECREMENT

the implementation of a particular algorithm, a higher-level language program also must provide the compiler with descriptive information that allows it to perform a variety of organizational tasks directly connected with the production of an executable object program. For example, the description of a variable (its name, together with the type of data to be stored in it) enables the compiler to allocate the proper amount of storage, associate its location with the variable's name, and set up any necessary data conversion mechanisms prior to the assignment of a value to that variable. (This description also defines the set of operations that are applicable to the element.) Similarly, the definition and description of a data file makes it possible for the compiler to establish a relationship between references to that file and a particular collection of data transmitted to or from a specific input/output device.

In most languages, this type of information is supplied through a series of special statements, which often are characterized as being "non-executable" (or more properly, declarative). Once defined, simple variables, arrays, files, and other items can be used throughout the program simply by alluding to their properties by use of their names,

To illustrate the type of information conveyed by declarative statements, consider the following Fortran program, which reads a number N and uses it to compute

$$Y = \sum_{X=1}^N X(1 + \sqrt{X}).$$

N and Y are printed with appropriate identification:

```
INTEGER N  
REAL X,Y  
READ (5,8) N  
Y = 0.0  
DO 6 I = 1 ,N  
  X = I  
6  Y = Y + X*(1.0+SQRT(X))  
  WRITE (6,16) N,Y  
8  FORMAT (12)  
16 FORMAT (4HN = ,I2,4X,4HY = ,E12.5)  
END
```

The first two declarative statements (underlined) define the three variables and instigate the necessary storage allocation; statement number 8 describes the form in which the input value will be found (a 2-digit integer punched in the first two columns of a card), and statement 16 provides formatting information for the output. Associations

with the respective **READ** and **WRITE** statements are provided by appropriate statement number references.

Many other languages provide similar declarative facilities which may be interspersed throughout the program. A notable exception is Cobol, in which the declarative facilities are much more formally structured: Each program must consist of four organizational divisions in a fixed order, the first three of which consist entirely of declarative statements.

S. V. POLLACK

DECREMENT

For article on related subject **see ADDRESS-ING.**

For article on related term **see INDEX REGISTER.**

The dictionary defines a decrement as a negative increment, but in a computing system, a decrement is a value that is subtracted from the contents of a register.

The term was used in the IBM 704 system and in successor IBM 700 and 7000 systems to denote a field that occurred in instructions to manipulate the index registers. Fig. 1 shows the format of these 36-bit instructions.

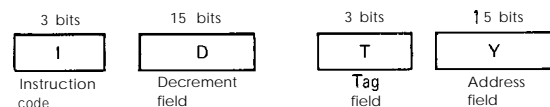


Fig. 1. Format of 36-bit instruction with decrement field.

For example, if the first three bits were 010, the instruction was **TXI** (Transfer on Index). The tag field specified an index register, and this instruction compared the decrement (i.e., the contents of the decrement field) with the contents of the index register. If the decrement D was smaller, the contents of the index register would be decreased (decremented) by the quantity D , and a transfer of control would occur to the address Y specified in the address field.

This class of computers, though very widely used, was almost unique in the use of subtracting rather than adding in connection with index registers. It was therefore appropriate to use decrements rather than increments in the index register with the modify-and-test instructions.

S. ROSEN

DELAY-LINE MEMORY, *See* ULTRASONIC MEMORY.

DELIMITER

For article on related subject see **PROCEDURE-ORIENTED LANGUAGES**.

A delimiter is an item of lexical information whose form and/or position in a source program denotes the boundary between adjacent syntactic components of that program; this term is also used for a program component whose value controls the number of iterations in a cyclic process.

As is true with natural language, the "meaning" and clarity of statements in higher-level programming languages often depend on the inclusion of explicit indicators that "punctuate" the statement. In the latter context, such signals are termed "delimiters." Since higher-level language statements must be processed by a compiler whose analytical and interpretive facilities must function without the equivalent of human cognition, it is necessary to equip programming languages with a fairly extensive variety of such delimiters, many of them highly specific. The most common of these, of course, is the blank space, whose function as a separator is self-explanatory. As a convenience to programmers, many compilers tolerate superfluous blanks between syntactic components.

Parentheses also represent a commonly used type of delimiter. One of their primary purposes in higher-level languages parallels traditional mathematical usage; i.e., to define the extent of a component in a computational expression. For example, the use of parentheses in the ordinary arithmetic expression

Is clearly paralleled by the equivalent in many higher-level languages. For example,

$$A + B * (C - 2 * D) \quad (2)$$

Many programming languages provide a relatively free physical format where there is no intrinsic association with a specific input medium, such as the punched card or teletypewriter. Consequently, there is no implicit correspondence in such languages between the end of a statement and the physical boundary of the medium, thus necessitating the use of explicit delimiters. The semicolon serves that purpose in PL/I and Algol, for example, so that a statement in those languages may be defined operationally as a string of characters between two semicolons. Similarly, the period delimits certain types of Cobol statements.

In block-oriented languages such as PL/I and Algol, special delimiters are provided to indicate the boundaries of major structural components. For instance, an arbitrarily long sequence of PL/I statements may be identified as a "compound" statement in certain contexts by using the delimiting statements **DO** and **END** at the beginning and end of the grouping, respectively. Similarly, a PL/I procedure is bracketed by **PROCEDURE** and **END** statements. (The **BEGIN** and **END** statements provide the same definition in Algol.)

In a somewhat different context, the term "delimiter" is used to refer to a numerical value (usually expressed as a constant, variable, or arithmetic expression) that controls the number of cycles in an iterative process. A very frequently used technique for implementing such a cycle is exemplified by the following sequence of PL/I statements:

```
Y = 0;
DO I = 1 TO 17;
Y = Y + I;
END;
```

In this construction, a variable *Y* is set (initialized) to 0, followed by a set of exactly 17 cycles. During each of these, the value accumulated thus far in *Y* is increased by adding the current value of *I* to it. As an integral part of this process, the value of *I* is increased by 1 each time through, so that after the seventeenth cycle has been completed, the final value of *Y* represents the sum of the first 17 integers. The **DO** statement contains the information required for the compiler to construct the cycle, and the item following the word **TO** (17 in this example) is the delimiter of the cycle.

$$A + B(C - 2D) \quad (1)$$

S. V. POLLACK

DESIGN, LOGICAL

DESIGN, LOGICAL. *see* LOGIC DESIGN.

DESK CALCULATOR. *See* CALCULATOR, DESK.

D ETAB. *See* DECISION TABLES: Languages.

DIAGNOSTICS

For articles on related subjects *see* **DEBUGGING; ERRORS; and MICROPROGRAMMING.**

Diagnostics are means of determining whether there are hardware faults in a computer or errors in user programs.

Hardware diagnostics are programs designed to determine whether the components of a computer are operating properly. Circuit components are electronically exercised individually and in groups with the intention of detecting failures. When a failure is detected, the location and identification of the faulty element is printed and the maintenance staff can take action to repair or replace the element.

Hardware diagnostic programs are run as part of a regular schedule of preventive maintenance and in the event of a failure. If a failure has occurred, the hardware diagnostic program may not operate properly, and therefore it may not be of use in determining the source of the difficulty.

Increasingly often hardware diagnostics take the form of microdiagnostics. A microdiagnostic program is a microprogram that tests a specific hardware component such as a bus or store location. Microdiagnostics often provide more accurate location of a fault than hardware diagnostics written in machine language because of the addressability of individual components under microprogramming. Furthermore, these diagnostic programs are so fast that preventive maintenance testing may be interspersed transparently with other processing. Microdiagnostics, consequently, have furthered the development of self-diagnosing and self-repairing computers.

Diagnostic messages in software refer to the error messages produced by compilers, utilities, and

system software. These messages are designed to give programmers an indication of where their programs are at fault. Diagnostic messages at compile time may only be warnings to the programmer, or they may indicate invalid syntax, which prohibits execution. In many systems a severity level indicator will be included in the diagnostic message.

Execution-time diagnostic messages are produced by the operating system or an execution-time monitor. These messages indicate attempts to perform illegal operations, such as dividing by zero, taking the square root of a negative number, illegal operation codes, illegal address references, and so on. The diagnostic message may or may not be followed by termination of the program.

Finally, application programs may produce diagnostics when erroneous control cards or data cards are read in. The creator of the application program has complete control over the nature of these diagnostic messages and the action to be taken.

B. SHNEIDERMAN

DIFFERENTIAL ANALYZER

For articles on related subjects *see* **ANALOG COMPUTERS; and DIGITAL COMPUTERS: Early.**

In a paper published in the *Journal of the Franklin Institution* in 1931, Vannevar Bush described a machine (Fig. 1) that had been constructed under his direction at M.I.T. for the purpose of solving ordinary differential equations. He christened the machine a *differential analyzer*. 'This was what would now be called an "analog" computer, and was based on the use of mechanical integrators that could be interconnected in any desired manner. The integrator was in essence a variable-speed gear, and took the form of a rotating horizontal disk on which a small knife-edged wheel rested. The wheel was driven by friction, and the gear ratio was altered by varying the distance of the wheel from the axis of rotation of the disk. The principle is illustrated in Fig. 2.

The use of mechanical integrators for solving differential equations had been suggested by Kelvin, and various special-purpose integrating devices were constructed at various times. Bush's differential analyzer was, however, the first device of sufficiently general application to meet a genuine need, and in

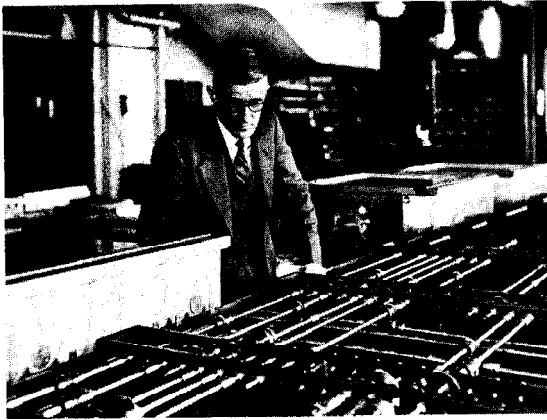


Fig. 1. Vannevar Bush shown with the M.I.T. differential analyzer.

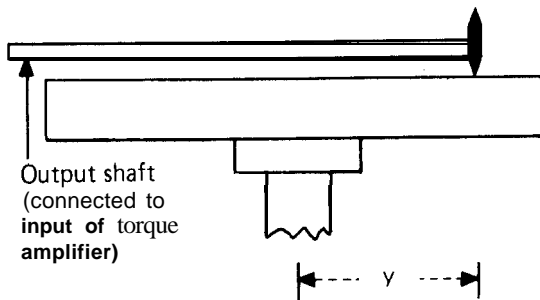


Fig. 2. Wheel and disk integrator. If the disk turns through an angle proportional to x , the output shaft turns through an angle proportional to $\int y \, dx$.

the period immediately before and during World War II quite a number of these devices were constructed. The one shown in Fig. 1 was installed at the Mathematical Laboratory, in Cambridge, England.

In order to make a practical device, it is necessary to have some means of amplifying the small amount of torque available from the rotating wheel. Bush used a torque amplifier, working on the principle of the ship's capstan, but adapting it for continuous rotation. Fig. 3 is taken from his report (1931) and sufficiently indicates the principle. The friction drums are rotated in opposite directions by a continuously running motor of sufficient power. When the input shaft is turned, one of the cords attached to the input arm begins to tighten on the friction drum round which it is wrapped. Which cord tightens depends on the direction of rotation of the input shaft. A very small tightening, and hence a very small tension in the end of the cord attached to the input arm, is sufficient, in view of the friction of the rotating drum, to produce a large tension in the end attached to the output arm. A small torque applied to the input shaft is thus capable of producing a much larger torque in the output shaft.

The integrators and torque amplifiers can be clearly seen in Fig. 4, together with the system of shafting used for effecting the connections. Changing the problem was a job for someone who did not mind getting his hands covered in oil. The output table on which the results were plotted directly in graphical form can also be seen in Fig. 4, which also shows a number of similar tables that were used for

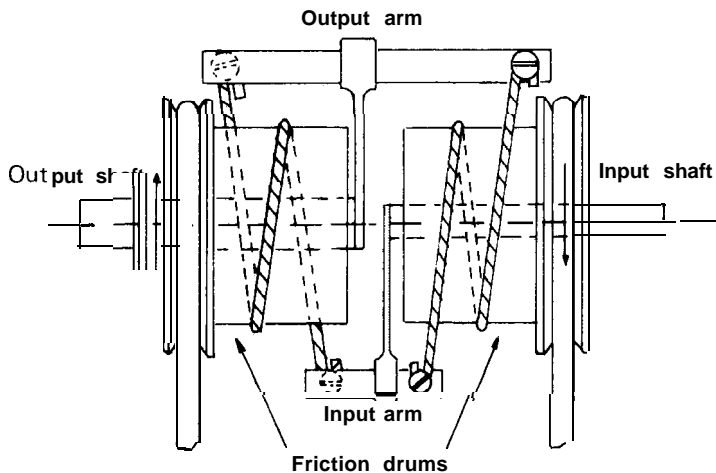
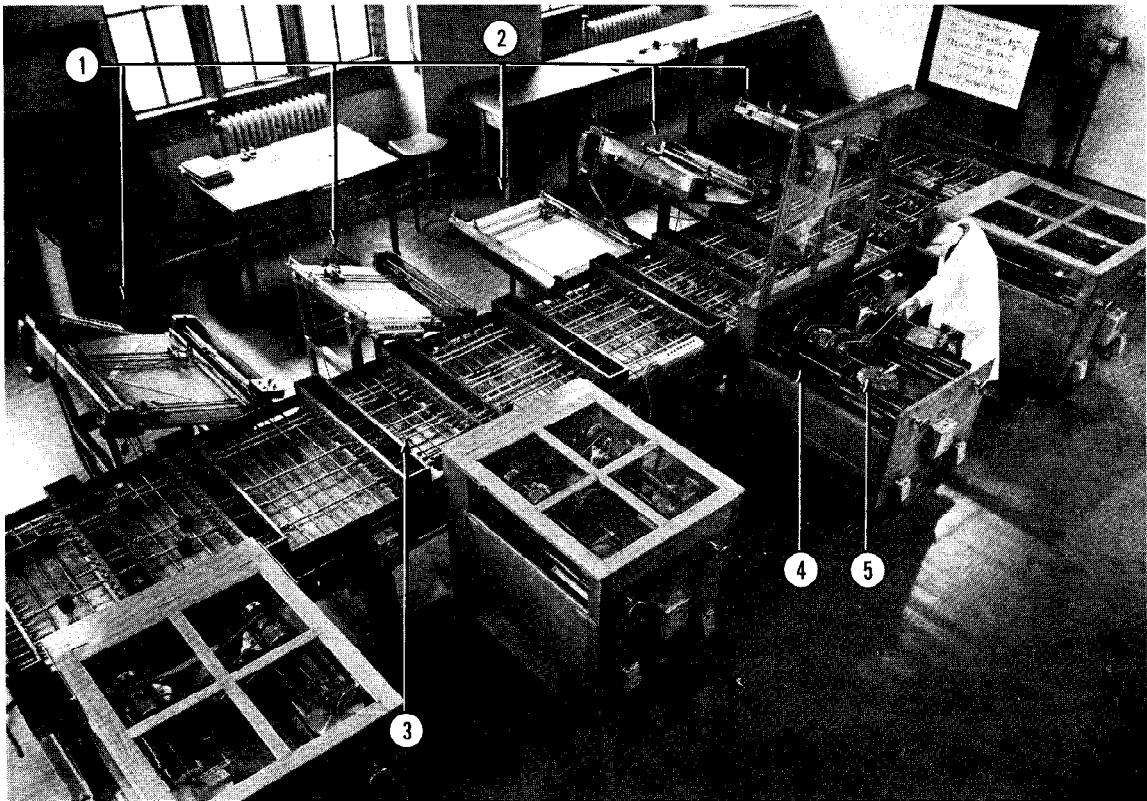


Fig. 3. Principle of torque amplifier. (Courtesy of *Journal of the Franklin Institute*.)

DIFFERENTIAL ANALYZER



- | | | |
|----------------|---|--------------------|
| 1 Input table | 3 Shafts and gears used for interconnection | 4 Torque amplifier |
| 2 Output table | | 5 Integrator disk |

Fig. 4. The differential analyzer system, showing integrators, torque amplifiers, and shafting.

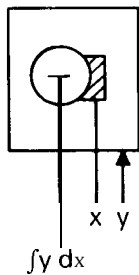


Fig. 5. Schematic notation for an integrator.

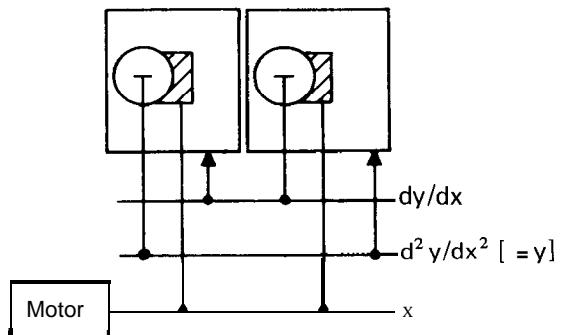


Fig. 6. Setup for solving the equation $d^2y/dx^2 = y$.

input, an operator being employed to turn a handle so that a cursor followed a curve. It is a comment on the primitive state of automatic control in the period in question that automatic curve-following devices were not provided until later. The accuracy attainable in a single integrator was about one part in three thousand, but of course a lower accuracy was to be expected in the solution.

Fig. 5 shows the notation that was used for an integrator and Fig. 6 shows how two integrators could be interconnected to solve a simple differential equation. It was not difficult to arrive at a diagram such as Fig. 6, even for a complicated equation, but working out the gear ratios required was a distinctly

tedious task calling for some experience, particularly as accuracy required that full use should be made of the available range of integrator motion.

In 1945, Bush and S. H. Caldwell described a new differential analyzer in which interconnection between the integrators was effected electrically instead of mechanically. However, during the decade that followed, competition from electronic analog computers and from digital computers began to build up, and although the new machine ran for a number of years at M.I.T., by 1955 the mechanical differential analyzer was already obsolete.

Digital Differential Analyzer. This device is based on the use of a *rate multiplier* as an integrator. In a rate multiplier a constant quantity y is held in a register and, on the receipt of an input pulse, is added to the number standing in an accumulator. If input pulses arrive at a rate R , overflow pulses will emerge from the most significant end of the accumulator at a rate proportional to yR . If y now varies and if input pulses arrive whenever a certain other variable x increases by δx , the number of output pulses emerging is proportional to $\sum y \delta x$ or, approximately, to $\int y dx$. Thus, the device serves as an integrator. Normally, δx is equal to one unit in the least significant place, and continuously updated values of the variable x can be obtained by feeding the pulses into an accumulator.

The first digital differential analyzer was the MADDIDA developed in 1949 at the Northrop Aircraft Corporation. It had 44 integrators implemented using a magnetic drum for storage, the addition being done serially. There were six tracks in all on the drum, one being used for synchronizing purposes. The problem was specified by writing an appropriate pattern of bits onto one of the tracks. Compared with the digital computers then being built, the MADDIDA was on an impressively small scale. It lost some of its simplicity, however, when adequate input and output devices were added, and in the end competition from general-purpose digital computers proved too much for it. The MADDIDA and its descendants did not, therefore, have the bright future in scientific computation that was predicted for them. However, digital differential analyzers of a simple kind continue to have a place in certain control applications,

REFERENCES

1931. Bush, V. J. *Frank. Inst.*, Vol. 212, p. 447.
 1945. Bush, V., and S. H. Caldwell. *J. Frank. Inst.*, Vol. 240, p. 255.

1947. Crank, J. *The Differential Analyser*. London: Longmans, Green and Co.

1962. Huskey, H. D., and G. A. Korn. *Computer Handbook*. New York: McGraw-Hill, pp. 19-14.

M. V. WILKES

DIGITAL COMPUTERS

GENERAL PRINCIPLES

For articles on related subjects see **ANALOG COMPUTERS**; **ARITHMETIC-LOGIC UNIT**; **HYBRID COMPUTERS**; **MEMORY**: Auxiliary; **MEMORY**: Main; and **SPECIAL-PURPOSE COMPUTERS**.

For articles on related terms see **CALCULATOR**, **DESK**; **CENTRAL PROCESSING UNIT**; and **FILES**.

The digital computer is a machine, a machine which will accept data and information presented to it in its required form; carry out arithmetic, transfer, and logical operations on this raw material; and then supply the required results in an acceptable form. The resulting information (output) produced by these operations is entirely dependent upon the accepted information (input). Thus, the production of correct and complete answers cannot be obtained unless correct and sufficient input data has been provided.

The sequence of the operations required to produce the desired output must be accurately determined and specified by a human-known as a "programmer"-who prepares a set of instructions-known as a "program"-that will automatically process the work from input to output.

Computer Characteristics. The main characteristics of the computer are identified as automatic, general purpose, electronic, and digital. Fig. 1 shows a computer installation at Oxford University.

AUTOMATIC. We assume that a machine is automatic if it works by itself without human intervention. But this is not entirely true. Computers are machines: They have no will of their own; they cannot start themselves; they cannot go out and find their own problems and how to solve them. They have to be instructed. They are, however, automatic in that once started on a job, they will carry on until



Fig. 1. ICL 1906A computer system at Oxford University.

it is finished, normally without human assistance.

A computer works from a program of *coded* instructions that specify exactly how a particular job is to be done. While the job is in progress, the program is *stored* in the computer, and the parts of the instructions are obeyed. As soon as one instruction is completed, the next is obeyed automatically.

By contrast a desk calculator is semiautomatic; somebody has to set up the numbers on a keyboard and press a button (add, subtract, multiply, or divide) to initiate each individual operation. It should be noted, however, that with one operation, some modern electronic desk calculators are able to perform quite complicated functions.

Because a computer does not need to stop between single operations, it can take full advantage of the high-speed components that enable it to add, subtract, and perform other individual operations in millionths of a second.

GENERAL PURPOSE. Computers (and desk calculators, though they are rarely described as such) are *general-purpose* machines. In other words, a

computer can do any job that its programmer can break down into suitable basic operations. Put a payroll program into a computer and you make it, for the time being, a special-purpose payroll machine. Replace the program by one for inverting a matrix, and you make the computer temporarily a special-purpose mathematical machine.

ELECTRONIC. The word “electronic” refers, of course, to the components of the machine. It is the nature of the electronic components that make possible the very high speeds of individual operations in modern computers. (“Electronic” is distinct from “electric” in that it suggests not only circuits and currents but also such components as transistors or their equivalent.) The history of electronic digital computers distinguishes three “generations,” defined by the nature of the electronic components most prevalent in each. Thus, the first generation made extensive use of vacuum tubes, the second generation used transistors, and the third generation uses integrated circuits.

Most computer users need no special knowledge

of electronics, and the major practical distinctions between the generations-as far as they are concerned-are the reductions in size for a given power and the rapid increases in speed.

DIGITAL. A computer may be either *digital* or analog. The two types do have some principles in common, but they employ different types of data representations and are in general suited to different kinds of work. Digital computers are so called because they work with numbers in the form of separate digits. More precisely, they work with information that is in digital or character form, including alphabetic and other symbols as well as numbers.

In a digital machine the data, whether numbers or letters or other symbols, is represented in digital form. An analog computer, on the other hand, may be said to deal with a "model" of the problem, in which the variables are represented by physical quantities such as angular position and voltage. The decimal numbers 136 and 435, for instance, might be

represented by by 1.36 volts and 4.35 volts. Using familiar devices, we could say that a slide rule is an analog device because numbers are represented by a linear length. The abacus, on the other hand, is a digital device because movable counters are used for calculating.

Digital computers differ from analog computers much as counting differs in principle from measuring. Both types of machines employ electric currents, or signals, but in the analog system a number is represented by the magnitude (e.g., voltage) of a signal, whereas in a digital computer it is not the magnitude of signals that is important, but rather the number of them, or their presence or absence in particular positions. Analog computers tend to be special-purpose machines designed for some specific scientific or technical application. They are frequently found useful in engineering design for such things as atomic power stations, chemical plants, and aircraft. In commercial and administrative data processing and for mathematical computation, we

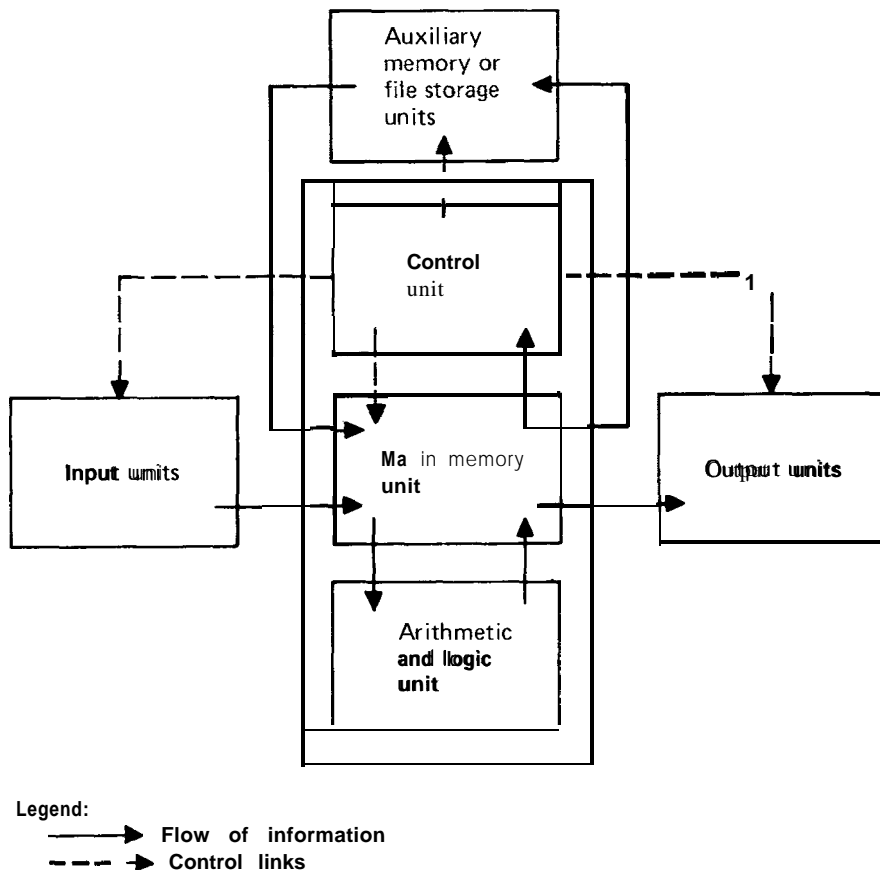


Fig. 2. Central processing unit. Grouping of computer components.

DIGITAL COMPUTERS

are concerned almost exclusively with digital computers.

Main Units. Only very rarely indeed does a computer have a unique fixed specification. Normally, it is better described as a computer system, consisting of a selection from a wide variety of units appropriate to meet a defined need. The principal groupings of these units commonly follow the pattern shown in Fig. 2 and are defined as follows:

1. *Input units.* An input unit accepts the data, the raw material that a computer uses, communicated from outside. It is the actual means by which information is converted into electronic pulses, which are then fed into the machine's memory.

2. *Control unit.* The directing force of the computer, the automatic operator, is the control unit. It provides the means of communication within the machine, by moving, advancing, or transferring information. It switches and integrates the various units into a whole system.

3. *Main memory unit.* All information is stored in the main memory unit and is "remembered" and made available to other units as required, under the direction of the control unit.

4. *Arithmetic and logic unit.* This unit performs the four arithmetic operations of add, subtract, multiply, and divide. By determining whether one number is larger than another, whether it is zero or whether it is a plus or minus, it is said to have logical abilities; i.e., it can make logical "predetermined" decisions. In addition, it can sometimes perform other strictly logical operations.

5. *Output units.* After information is processed, an output unit communicates it to the outside world or returns it to memory for later use by the computer. When needed, the results are recalled from memory under the direction of the control unit and presented by the output units in an appropriate form. A wide variety of output devices is available.

6. *File storage units.* These units store information required for reference.

We will now consider each of these main units in some detail.

INPUT. The input function is fulfilled by the individual *input devices*, which "read" information into the memory from *media* such as punched cards, paper tape, or special documents that can be read directly. Other devices enable us to communicate directly with the computer through a typewriter-like keyboard, or permit direct communication of data transmitted from a distance over telephone lines.

Only electronic components can give the computer the internal speed of operation that is its great advantage, so its methods of accepting and presenting information are necessarily electronic in nature. But information available outside the machine is perforce initially in the form of written or printed words and figures because these are the only forms interpretable by humans.

The conversion of information from written form to computer form is usually done in two stages (Fig. 3). First, the information is punched into cards or paper tape, where it is represented by patterns of holes punched according to a standard code. At this stage the information may be viewed as being in an intermediate "physical" form. Special electro-mechanical devices known as card readers and paper tape readers are able to "read" the cards or tape. An input device detects where holes are present and passes corresponding electrical "pulses" into the internal part of the computer, thus completing the two-stage conversion. The value of a hole in a card (Fig. 4) or tape is dependent on the row it appears in and its meaning on the column it occupies, reading across the **card**. Combinations of holes in a single column represent alphabetic characters. A coded line of holes represents one alphabetic character or numeric digit .

Some devices, known as "document readers," are capable of directly reading letters or numbers printed on paper in suitable type fonts. The vertical bars or letters along the bottoms of checks are an example of one such font. The characters are printed in a magnetized ink, and when they pass the reading mechanism of the input unit, the shape of the magnetic field that is produced is assessed by the reader and the character is identified. Other document readers are capable of accepting type fonts that appear more normal to the human eye.

An important group of input devices are those based on modified electric typewriters or Teletype machines. These permit data to be communicated directly into the memory of the machine. As yet, this is not a popular method of inputting large volumes of data, but it is frequently used for interrogating the computer or, in the special case of the computer console typewriter, for direct communication between operator and machine.

For large volumes of input, there is a trend from the long-established punched card or paper tape toward magnetic tape (Fig. 5) as the recording medium. For example, a group of keyboards may be connected to a special-purpose ancillary computer that records the data in the form of tiny magnetized areas on the tape's surface. The coding is similar to

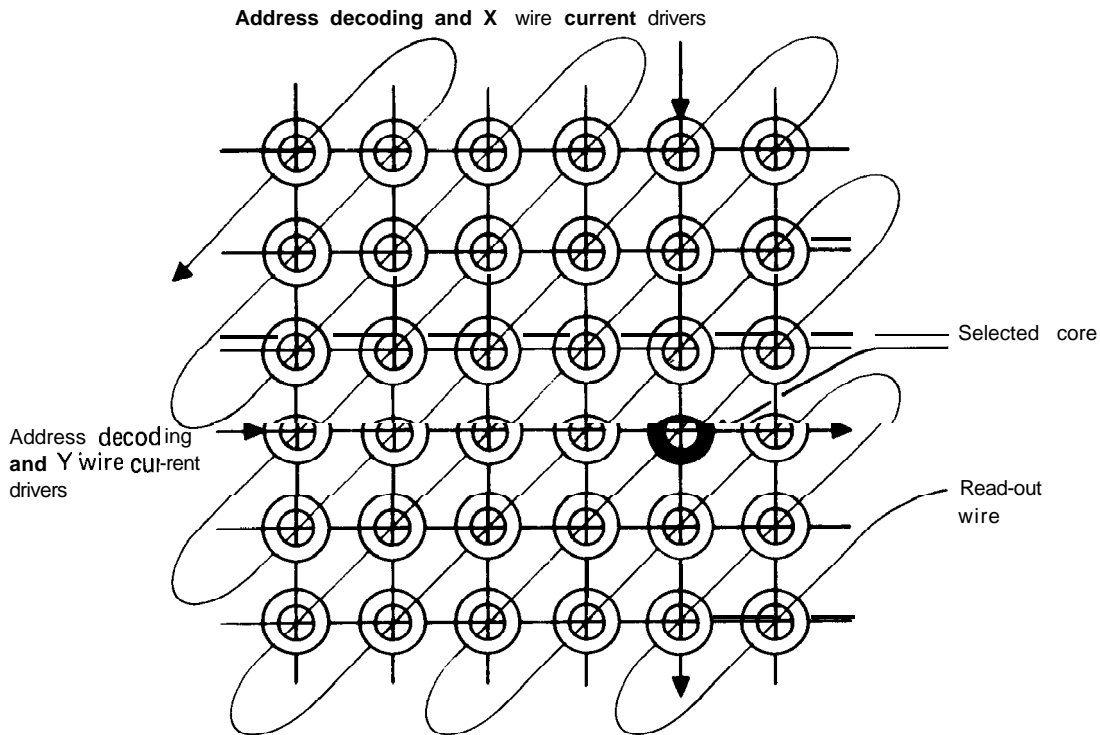


Fig. 6. Core storage.

an output unit.

The access time to data in the internal memory is important. The memory is a vital crossroad in processing, and the very high-speed arithmetic facilities of the machine demand virtually instantaneous access to data. Typically, memories can supply requested data in an incredibly short time, measured in microseconds (millionths of a second) or nanoseconds (thousand-millionths of a second); in fact, their speed is usually from 8 μ s down to 200 ns.

The second use of the memory is to hold all the instructions of the program required to carry out a job. These instructions are normally coded in numeric form and can be read into the memory from punched cards, magnetic tape, or any other input medium. They remain in memory indefinitely unless they are deliberately erased.

The most common memory device is the magnetic core. This consists of small ferrite rings that can be magnetized in either of two directions, clockwise or counterclockwise. One state of magnetization denotes 0 and the other denotes 1; hence, each core can represent a binary digit, or "bit." A group of these bits (or cores), using a suitable form of coding (just as in the Morse code), can represent

decimal digits or alphabetic characters. The cores are strung on a rectangular mesh, or matrix, of wires. Each core is located at the intersection of a vertical and a horizontal wire, and a third wire (the read-out wire) passes through all the cores (Fig. 6).

A three-dimensional block of core storage is made up of a number of two-dimensional planes, arranged in such a way that the group of bits representing a character can be written magnetically and later read when required.

Core storage works on the principle of permanent electromagnetism. If sufficient current is passed along a wire running through a core, magnetism is induced in a sense (direction) related to the direction of the current (clockwise or counterclockwise). A core will retain its magnetism indefinitely until a reverse current passes along the wire, when the sense of magnetization will be reversed. By sending a pulse of current along the appropriate wires for a given address, the data or instruction held in that address will be transmitted along the "read" wires for routing to another part of the system.

2. *The arithmetic and logic unit.* This is obviously the part of the machine simplest to understand. It is where actual arithmetic operations are carried out. It

is quite common to find a machine that can add a pair of eight-digit numbers in about ten millionths of a second, and there are models that can do the job in one millionth of a second or even much less!

The term "logic" is used here to describe a nonarithmetic facility of the unit; i.e., its ability to differentiate between positive and negative numbers, and as a result, to take alternative paths in the program. A simple example will easily illustrate its value.

In stock control it is usual to compare a newly calculated stock balance with the preset minimum or danger level, to determine if reordering is necessary. If the "minimum" is subtracted from the "balance," then a positive (excess) result indicates that all is well; a negative result (a shortage) shows a need to reorder. In this latter case only, we can arrange for the machine to jump to a part of the program that prints out reordering information on the printer for management action. If all is well, we need print nothing—one way in which the computer itself can reduce paperwork. This apparently simple facility is of fundamental significance, and a typical program of a few thousand instructions will contain many of the "test and jump to another phase of the program if negative" type of instruction.

The arithmetic and logic unit consists of one or a number of registers (each made up of electronic circuits), which may be termed "accumulators." To add a number stored in address 113 of the memory

to that stored in address 207, first the contents of address 113 are read into the accumulator and then the contents of address 207 are added into the accumulator. The answer is then read out to another address in memory, thereby leaving the accumulator free for the next operation.

3. *The control unit.* This part of the central processor functions so as to cause the whole machine to operate according to the instructions in the program. Instructions are normally transferred sequentially from the memory to the control unit, where each instruction is interpreted and the appropriate circuits are activated to "execute" the instruction. This strict sequence is broken, for example, when a "test and jump" type of instruction occurs and produces an exceptional result. There is then a "transfer of control" to a program step in a different part of the program, from which the sequential pattern continues until again broken.

The control unit of a computer contains special circuits known as "microprograms." One of these corresponds to each type of elementary operation (and therefore to each type of instruction) in the computer repertoire. It is by the inclusion of a suitable microprogram that a given operation is "built" into the computer.

OUTPUT. The output devices of the computer enable it to communicate results to the outside world. Output form falls into two main categories:

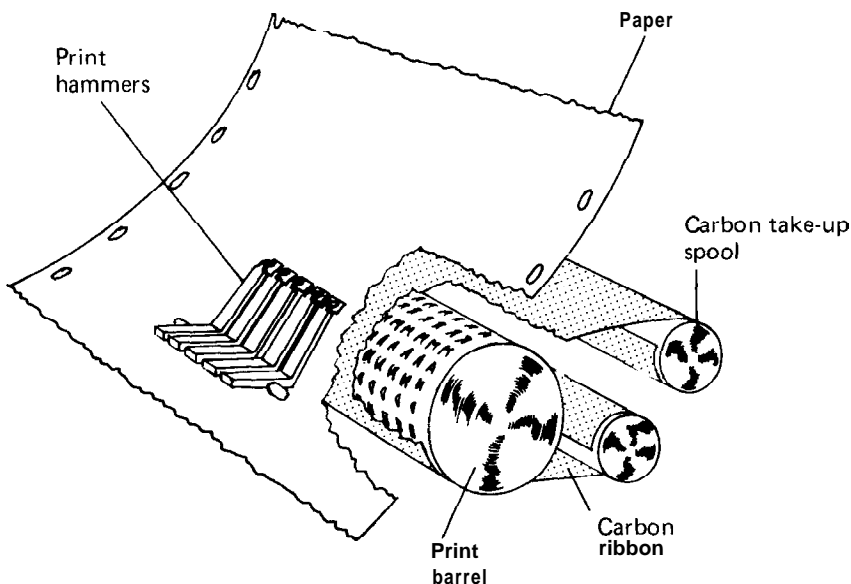


Fig. 7. A line printer (barrel type).

DIGITAL COMPUTERS

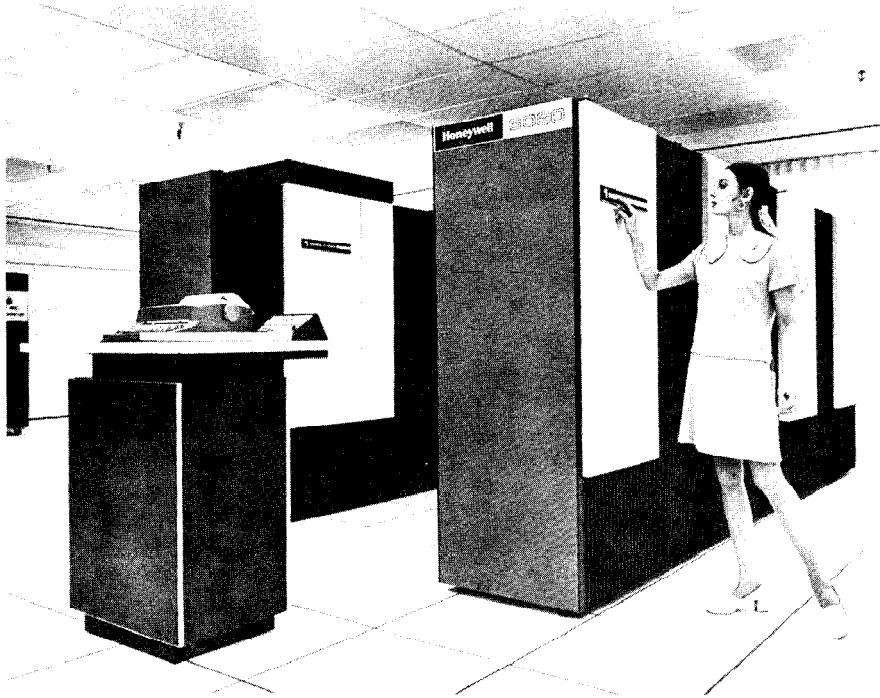


Fig. 8. Console typewriter for Honeywell 6080 computer system.



Fig. 9. Honeywell 775 cathode-ray tube display terminal.

1. Readily handled and understood by human beings (printers and display units).
2. Intended for further processing by machine (tapes and cards).

The first group contains a number of types of devices. The most obvious of these are printers, designed to produce results in the form of printing on paper.

1. *Printers and display units.* Most printers in current use operate on the same basic principle as the typewriter, in which a character is printed through an impact of a type face on an inked ribbon traversing the paper. There are, however, some printers that generate and print characters electronically.

Where the volume of printing is large, it is usual to use a "line printer," one capable of producing a whole line of print at a time (usually from 120 to 160 characters). Such printers are capable of quite high speeds, typically up to 20 or more lines per second on a continuously fed roll of stationery. It is essential, of course, to have excellent paper-handling facilities to keep pace with such speeds. Such line printers (Fig. 7) are ideal for applications requiring voluminous end-results such as payrolls, or invoices, or inventory listings, but are too often used to print more than necessary. Typewriter-like devices (Fig. 8) are extensively used for such low-volume printed output, and their keyboards permit them to double as input units as well. These devices may be situated

close to the computer or at a remote point, receiving the output messages over a telephone line,

A related device, which is becoming increasingly common, is the video-display unit (Fig. 9). This has a keyboard like a typewriter, but the printing mechanism is replaced by a television-like tube on which letters or digits can be projected. Compared with the typewriter, they have the advantage of displaying a large amount of information at once (often several hundred characters), and a fresh display of additional information can be generated very rapidly. On the other hand, they **cannot** produce "hard copy" or a permanent record. Therefore, they are best used in circumstances where an operator needs to examine a quantity of output information that does not have to be printed.

Video-display units can also be used to display information in graphical form and also diagrams of moderate accuracy. Associated input techniques using a **lightpen** (a device that effectively draws lines electronically on the face of the tube) manipulate changes or additions to drawings and diagrams. Devices such as the **lightpen** are increasingly being used for computer-aided design in such fields as car body or electric-circuit design.

Where a permanent record of a graph or drawing is required, or where greater accuracy is needed, a graph-plotter (Fig. 10) can be attached to the computer and can produce intricate drawings on paper.

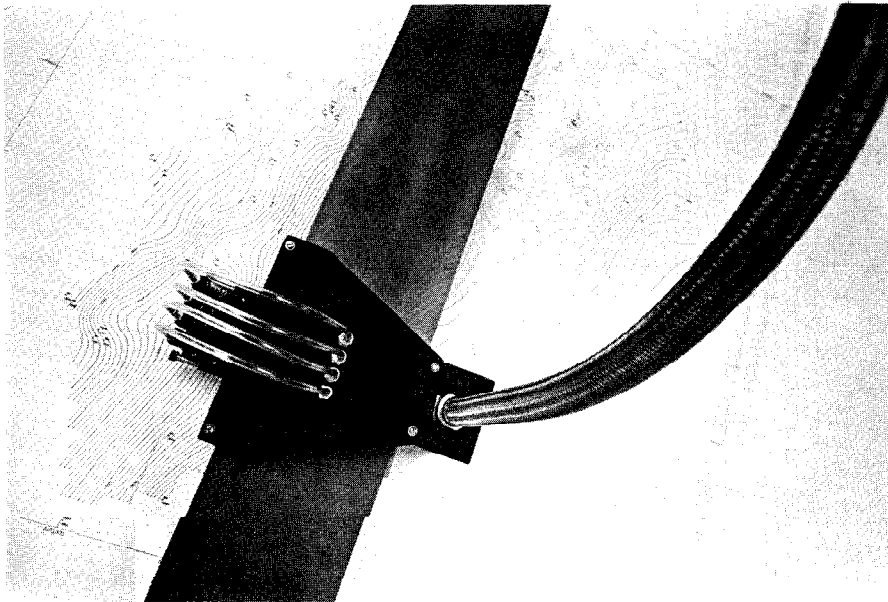


Fig. 10. Map being plotted by a flatbed plotter.

DIGITAL COMPUTERS

2. Tape and card media This second group of output devices produces results in a form for further processing. These include paper tape and card punches. Paper tape can be used for transmission of results over a teleprinter network or as the instructing medium for a numerically controlled machine tool. Punched cards can be interpreted (i.e., the characters represented by the holes can also be printed on the face of the card) and used as documents in their own right; for example, as job tickets or checks. They then provide a means of automatic repetitive input without key punching. This is an example of the "turnaround" document; i.e., a document produced by the computer, readable by a human being, and providing an automatic re-input medium to the machine. With the increasing availability of document readers, output documents printed in an appropriate type face can subsequently be used for input and also serve as "turnaround" documents.

Output data may also be recorded on magnetic tape and used for much the same purposes as paper tape. The principal differences, apart from the method of recording; are that magnetic tape operates at much higher speeds than the paper variety and is also much more expensive. A much more important role of magnetic tape is its role in providing a connecting link for carrying forward the results from one process which are to be the raw material for a subsequent process. Process A, for example, may well produce payroll information that will be subsequently used as input to Process B for a labor cost analysis. A special and vitally important aspect of this type of use is dealt with in the next section, "File Storage."

FILE STORAGE (OR AUXILIARY MEMORY). There are relatively few applications of computers, particularly in the field of business data processing, where the only input is fresh, raw data. For example, in inventory control the new data consists of stock issues and receipts, but data in file storage indicates the number of items left in stock calculated at last inventory and the average value of that stock. **These** files include more static information about each item, such as its name, dimensions, batch order, quantity, and supplier.

Thus, the computer must also have its "filing cabinets" (albeit electronic ones) if it is to be used in business applications. It is obvious that such file storage units must act as input and output units to the computer, and as they are of special and fundamental importance in these applications, they are treated here separately.

The three most important factors relating to any

filing system, whether manual or electronic, are (1) its total capacity, (2) the speed of access to required information-i.e., how long it takes to find what is needed, and (3) the cost per unit of data. The two most commonly used media for holding computer files are magnetic tapes and magnetic disks. Each system has its advantages and disadvantages, and the choice depends on particular circumstances; many installations indeed use both.

Storage Media

DATA ON TAPES OR DISKS. The different facilities provided by tape and disk give rise to two different philosophies of data processing: "serial access" or "batch" processing using magnetic tape, and "direct access" processing using magnetic disks.

Magnetic Tape. Records are held on tape in a serial fashion in a fashion analogous to a tape recorder. Thus, if the records are in no particular sequence, the user is forced to hunt backward and forward along the tape for any desired record. Although tape moves quite quickly on a computer tape unit (**up** to 12 ft or so per second), a reel of tape is usually 2,400 ft long and contains many thousands of records. (A 1-in. length of tape can accommodate up to 800 or more characters.) Thus, minutes could easily elapse between the location of succeeding required records. It is obvious, therefore, that the records in a magnetic tape file must be held in some predetermined sequence, such as employee-number or customer-number sequence, and that the new data being input in order to bring the file up to date (to "update" the file) must also be in the same sequence.

When used for file storage, magnetic tape units (Fig. 11) are generally operated in pairs: one carrying the brought-forward or current file, which is read by the machine; the other, a new carry-forward or updated file recorded or "written" by the machine, which in turn will become the brought-forward file when the job is next run. (Unaffected items on the brought-forward tape are obviously copied unchanged onto the carried-forward tape.)

Magnetic tape units (or decks, or transports, or stations) differ widely in performance (and price!), with reading and writing speeds varying from about 10,000 characters per second, through a common speed of about 60,000 characters per second, to a maximum speed of over 300,000 characters per second. Obviously, a tape system offers unlimited file capacity at low cost per record, but it is not an acceptable medium when immediate random access is required to every item in the file. A typical magnetic tape deck arrangement is shown in Fig. 12.

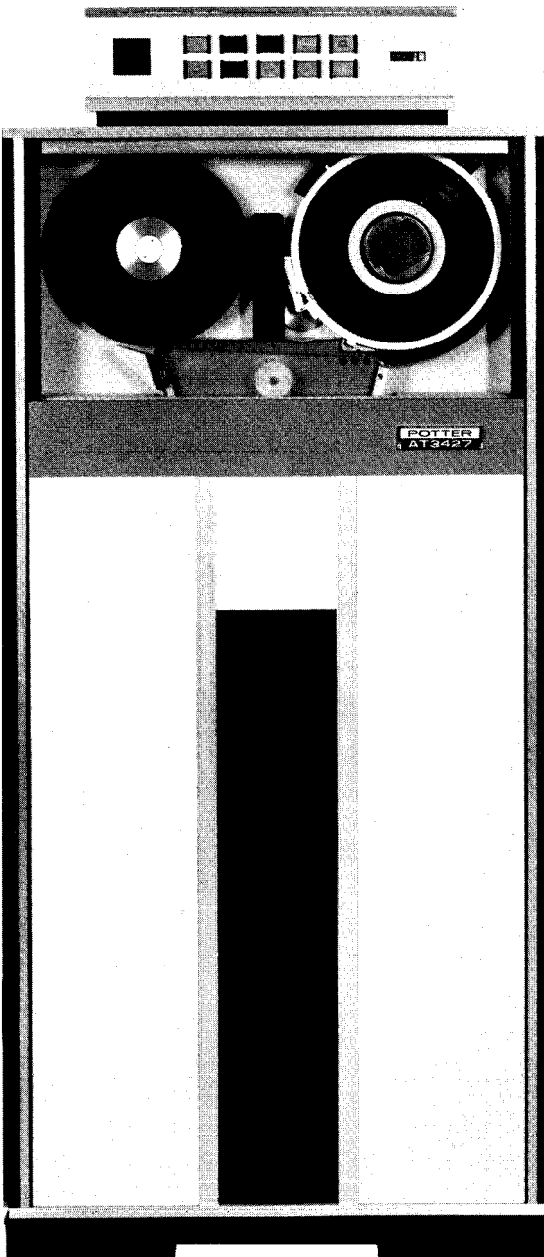


Fig. 11. Potter magnetic tape unit.

Magnetic Disks. There are two basic forms of the disk unit: In one the disks are usually large (over 3 ft in diameter) and are not removable from the unit; in the other, the "exchangeable disk," or "disk pack," system (Fig. 13) the disks are smaller (about 20 in. in diameter) and are demountable, usually in sets of 6 to 12. This is analogous to putting six phonograph records on a player that is equipped

with one pickup for each of the surfaces of the set. (Computers can "play" their disks on both sides without inverting them!)

Unlike the phonograph record, which has one track spiraling from the periphery toward the center, the computer disk has a large number of concentric tracks on its surface, each capable of storing about 4,080 characters. The pickup, or "recording head" can be moved radially across the disk surface to the desired track at very high speed. The total capacity of a set of six exchangeable disks (Fig. 14) is usually between four and eight million characters, and the recording head can move from one track to any other so quickly that direct access to any record at random can be obtained in about one-tenth of a second, or even less in some cases.

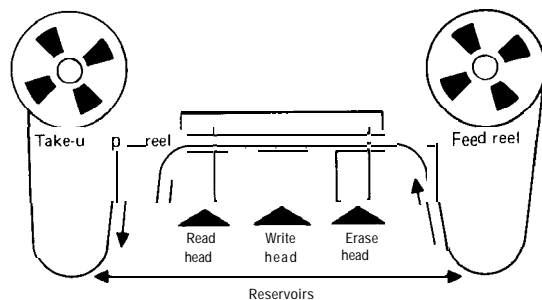


Fig. 12. Magnetic tape deck arrangement.

Thus, disks offer the facility of processing data in random sequence without any undue delay in searching for a required item. However, their capacity is much more limited than magnetic tape and their cost is higher. In addition, the fact that essentially random access is possible means that, in contrast to magnetic tapes, data can be corrected or changed and the modified record can be put back on the disk in the same place that it originally occupied. One disadvantage is that the modified data could create an overflow problem, and then a way must be found to correct it.

There are many facets to the problem of choosing between tape and disk for file storage. The simplest basis for choosing one over the other is that tape methods are in general cheaper, whereas disk systems offer greater speed and flexibility in the processing method, especially where the files must be frequently interrogated. Obviously, the choice depends on the application.

MAGNETIC DRUMS. Another medium (although less important than tape or disk) used for file storage is the magnetic drum. Historically, it is interesting to

DIGITAL COMPUTERS



Fig. 13. Honeywell 273 magnetic disk pack drive unit. Each pack has 20 surfaces and stores 18.4 million characters.

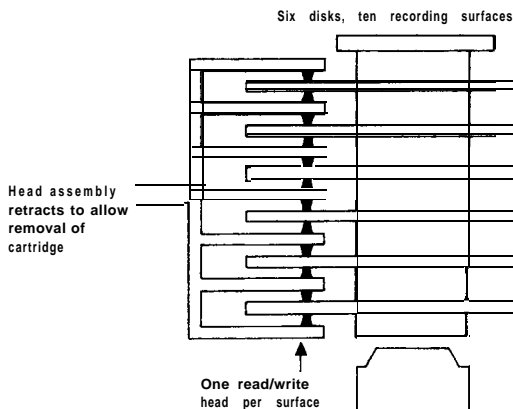


Fig. 14. Position of disk-pack components.

note that in early computers a magnetic drum frequently provided the main memory. Its speed of access, however, was such that other processing units were frequently kept waiting for instructions and data, so it was replaced in favor by magnetic core storage.

However, this inherent limitation does not affect its use as a file store, now that drums of large

capacity (up to 8 million characters and more) have been developed.

On a magnetic drum the curved surface of a rapidly rotating cylinder is the recording medium. There are a large number of magnetic heads, each of which can read and write data on the drum. Each head is associated with a specific recording track that extends around the circumference of the drum. In many respects the principles of operation and use of the magnetic drum are similar to those for the magnetic disk, but there are the following important differences:

1. Since there is usually a recording head for every track, no time is lost in the physical movement of heads to a required track.
2. A typical drum rotates about three or four times as fast as a magnetic disk system, which means that less time is lost waiting for the required data to come around to the recording head. Since each track has its own recording head, the fast rotation means that drums have a much shorter access time than disks.
3. Drums are more expensive per record stored than are disks. Moreover, they are permanently attached and are not exchangeable.

GENERAL. There are a variety of other file storage devices in addition to the more common ones discussed above. The struggle in many business applications is always to accommodate larger and larger files with an acceptable access time without paying too high a price.

G. J. MORRIS

DIGITAL COMPUTERS: HISTORY CONTEMPORARY AND FUTURE

For articles on related subjects see **DIGITAL COMPUTERS: Origins of, and Early; GENERATIONS, COMPUTER; MANUFACTURERS, COMPUTER; MICROCOMPUTER; and MINI-COMPUTERS.**

For articles on related terms see **CONTROL DATA CORPORATION 6000 SERIES; IBM 360-370 SERIES; INTEGRATED CIRCUITRY; LIVERMORE AUTOMATIC RESEARCH COMPUTER (LARC); STRETCH; and UNIVAC I.**

The development of digital computers can be conveniently categorized as successive generations.

The First Generation. The modern history of computing starts with the invention of the stored program computer. The first generation of electronic computers is characterized by the use of vacuum tubes as active elements. In the first generation a number of storage media were tried for reasons of economy and reliability and the early computers can be classified according to the nature of their main memory system.

MERCURY DELAY LINE STORAGE. Although mercury delay lines were important in a number of early research computers, UNIVAC I was the only computer delivered commercially that used this type of memory. The first UNIVAC I was delivered on June 14, 1951, several years ahead of the delivery of any competitive system. UNIVAC designers felt at that time that mercury-delay line memory was the only memory available that could provide adequate reliability at reasonably high speed. Average access time to the 1,000-word main memory was on the order of 500 μ s. UNIVAC was a completely serial machine with duplicate arithmetic and control circuitry for detection of errors. The relatively low speed was partially compensated for by the use of

minimum access-time coding and other sophisticated software devices. UNIVAC I had a number of features that did not become generally available on other computers until years later. These included a buffered tape system that could read tapes both forward and backward.

ELECTROSTATIC STORAGE. Most of the successful electrostatic storage systems were based on the Williams tube developed at Manchester University in England. Typical memories had from 1,000 to 4,000 words with random access times of from 10 to 50 μ s. There is still some question as to whether these memories ever achieved adequate reliability. IBM's 701 (Fig. 1), of which 18 were delivered between 1953 and 1956, was the most important scientific computer that used this type of memory. Remington Rand offered the only competitive computer in this field, the 1103, which it obtained when it absorbed Engineering Research Associates.

IBM used electrostatic storage on its 702, which it started delivering for commercial data processing in 1955. The 702 was the prototype of many later character-oriented computers, but for a number of reasons, including memory problems, it was soon



Fig.1. An IBM 701 system.

DIGITAL COMPUTERS

withdrawn from the market.

MAGNETIC DRUM STORAGE. Prototype magnetic drum computers included the Harvard Mark III and the ERA 1101. The magnetic drum provided a large amount of slow memory at relatively low cost. The IBM 700 series and the UNIVAC 1103 series used drums as peripheral storage.

Typical drum-storage systems used a drum that rotated at about 3,600 rpm, providing average random access times of about 17 ms. This seems incredibly slow for a main memory by modern standards, but it was the only way to get moderately priced memory in any quantity in the early 1950s. The development of cheap, reliable magnetic drum systems made it possible for many companies to enter the computer field with a rather modest investment. There were literally dozens of magnetic drum computers of varying capacity that were the small-to-medium sized computers of the first generation. Only a few of those that are historically most important will be mentioned here.

Among the magnetic drum computers delivered as early as 1953 were the CADAC 100 series and the Consolidated Engineering Corporation 200 series. The CADAC was produced by the Computer Research Corporation, which was absorbed by National Cash Register Corporation and became the forerunner of other computers in the NCR line. Consolidated Engineering spun off the ElectroData Corporation, which marketed such computers as the

Datatron 203, 204, and 205. ElectroData was absorbed by the Burroughs Corporation and became an important part of Burroughs' computer activity.

IBM's 650 (Fig. 2) was introduced a bit later and soon became the most widely used of all first-generation computers. Many hundreds were delivered between 1955 and 1959. The 650 was somewhat faster than most other magnetic drum computers, but the chief reason for its great success was its well-integrated punched-card input and output and its adaptability to existing punched-card systems.

Remington Rand introduced a perhaps too ambitious UNIVAC file computer, a drum-and-tape based data processing system that was not a commercial success. Toward the end of the first generation, the company introduced the UNIVAC 80 and 90, which were reasonably effective competitors for the IBM 650 systems. The use of solid-state components should perhaps place these computers in the second generation, but the solid-state components in the UNIVAC 80 and 90 were magnetic amplifiers, not transistors. At best, these computers belong in a transitional stage between generations.

MAGNETIC CORE MEMORIES. By 1953, both M.I.T. and RCA had developed working models of coincident-current magnetic core memories. Subsequent litigation awarded patent rights to the group at M.I.T., which made the memory design available to the computer industry. RCA developed the Bizmac,



Fig. 2. An IBM 650 system.



Fig. 3. Datamatic 1000 computer installed at Michigan Blue Cross-Blue Shield, Detroit.

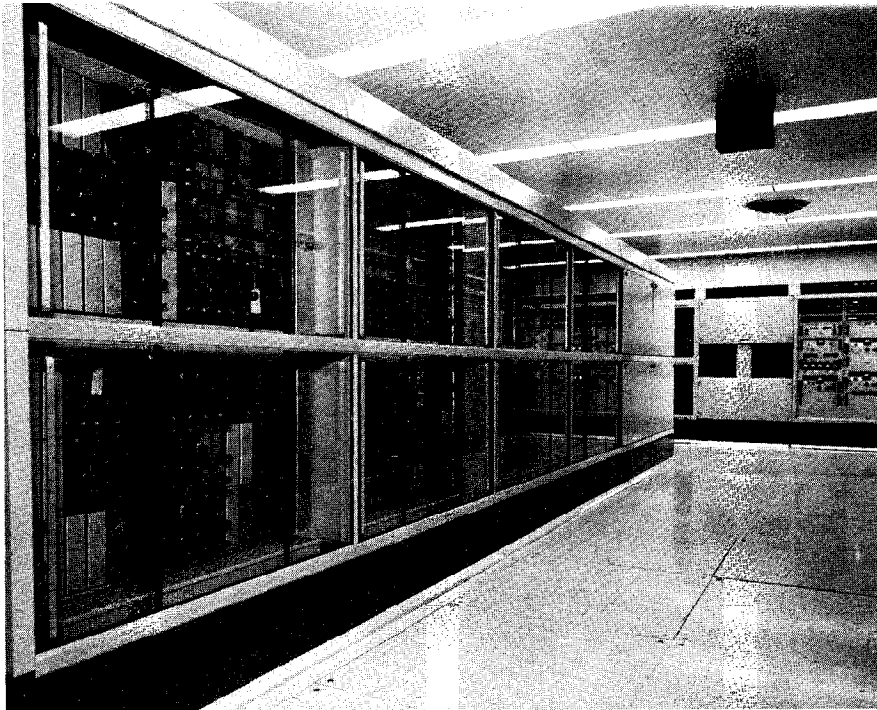


Fig. 4. Vacuum tube racks of the central processor of the Datamatic 1000.

DIGITAL COMPUTERS

a very ambitious commercial data processing system that was unsuccessful for a number of reasons, of which the most obvious was its failure to adequately exploit the capabilities of the magnetic core memory.

IBM moved quickly to adapt the core memory technology to both its scientific and its commercial computer lines, and in 1956 started deliveries on the 704—a very powerful successor to the 701—and the 705, a viable successor to the faltering 702. Core memory was faster and more reliable than the cathode-ray tube memories that were replaced. The hardware floating-point arithmetic and index registers on the 704 and the logical changes that permitted the 705 to work with groups of characters in multiple accumulators, coupled with the development of improved input/output and peripheral devices, made these computers orders of magnitude more powerful than their predecessors.

Remington **Rand's** UNIVAC division also quickly absorbed the new magnetic core memory technology and produced the UNIVAC 11, which was a compatible extension of UNIVAC I as an upgrade for its data processing customers. The 1103A was the magnetic core upgrade of the Univac Scientific Computer (the 1103), and optional floating-point and interrupt-handling hardware were soon added. The first UNIVAC I had been used to process the voluminous data collected in the 1950 Census. For the 1960 Census, UNIVAC added buffered input/output capacity to its scientific computer, which thus became the UNIVAC 1105. IBM added data channels to provide buffering capabilities to its 704 line, and introduced the 709 just as the first generation was coming to an end.

Other companies were quick to jump on the magnetic core bandwagon. Datamatic Corporation, which later became the computer division of Honeywell, produced a very large computer, the Datamatic 1000 (Figs. 3 and 4), which was used in a very few large data processing applications.

Burroughs produced its 220 system, a medium-size core memory machine that was much more powerful than competitive drum machines, but introduced it too late to have much impact, since its major competition was to come from second-generation machines.

The Second Generation. The transistor was invented in 1948, and the advantages of transistors over vacuum tubes for computer applications were recognized almost immediately. There were many technological and production problems that had to be worked out, and it was 1959 before transistorized computers were delivered in any quan-

tity. That year marks the beginning of the second computer generation, in which transistors completely replaced vacuum tubes as the active components of digital computers. All second-generation computers used magnetic core storage systems for main memory. Some of them used magnetic drums and disks in addition to magnetic tapes for auxiliary storage.

LARGE SCIENTIFIC COMPUTERS. Philco Corporation engineers developed the first transistors suitable for really high speed computers. Philco decided to enter the computer field with its own large-scale Transac S-2000 systems, and had moderate success with its Model 211 and later the more powerful 212, but could not generate enough momentum to carry it into the third generation. Philco withdrew from the general-purpose computer field in 1964.

UNIVAC developed one of the first successful, large-scale transistorized computers for military applications, the UNIVAC M460. A group of UNIVAC employees left the company and set up Control Data Corporation, which used the new transistor technology in their 1604 (Fig. 5) computer. The 1604 was followed by a more powerful 3000 series, the 3600 and later the 3800. Control Data established itself as a major supplier of large computers and retained and expanded its position in the third generation.

The possibility of using transistors in very large numbers to produce large and powerful computers was attractive to the Atomic Energy Commission, which sponsored two of the early second-generation projects, Stretch (the IBM 7030) at IBM and Larc at UNIVAC. Both companies tried to market the resulting computers in the early 1960s, but both were unsuccessful because the rapid progress of technology made these early giant computers uneconomical.

IBM produced the 709TX system in 1959 in response to the demand for a transistorized computer for the Ballistic Missile Early Warning System. The 709TX became the 7090, a compatible extension of the first generation 709, designed to run at more than five times the speed of the 709. The 7090, later upgraded to the 7094, dominated the scientific computer market in the period X960-1 964. A similar but slightly less powerful series, the 7040 and 7044 were introduced in 1962-1963. Combinations of the 7040 and 7090 series machines with disk storage formed the direct-coupled systems that were popular from 1964 to 1966 and provided a partial hardware and software prototype of some IBM third-generation systems.

The UNIVAC 1107 appeared late in the second generation and served mainly as a prototype for

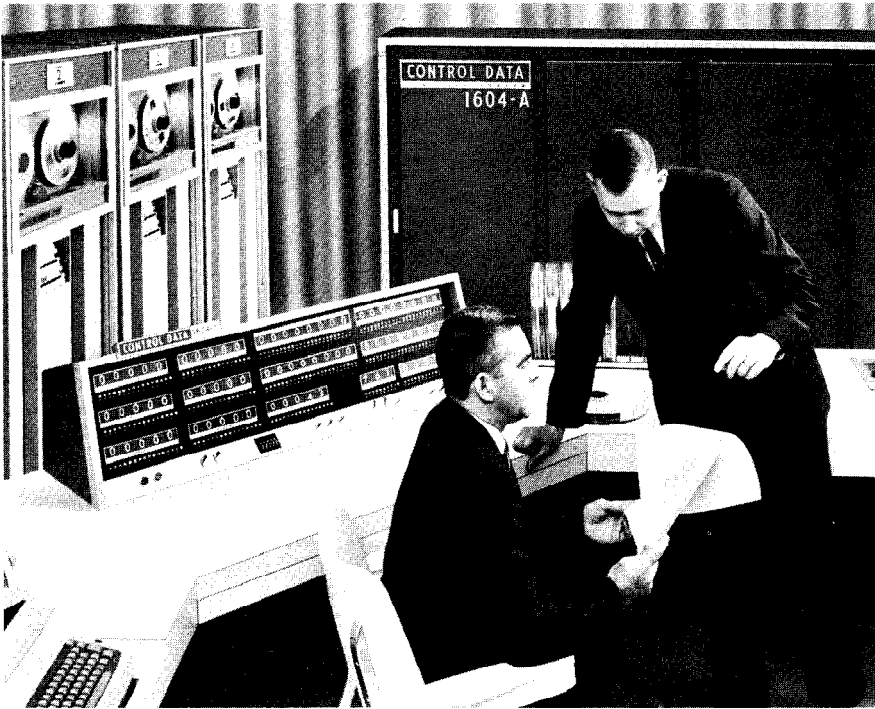


Fig. 5. CDC 1604-A computer.



Fig. 6. NCR 315 system.

DIGITAL COMPUTERS

the more successful 1108 model in the third generation.

DATA PROCESSING COMPUTERS. Some of the earliest second-generation computers were medium-scale data processing systems. National Cash Register was almost too early with its 304, a joint effort with General Electric. A more successful NCR 3 15 (Fig. 6) system was introduced in 1962. This system featured an interesting magnetic card cartridge auxiliary memory, CRAM.

The RCA 501 was another of the very early transistorized machines, but had limited performance. It featured one of the very first Cobol compilers. A much more powerful 601 introduced interesting microprogramming features. It was designed primarily for the scientific field, but was not competitive. RCA had most success with its small 301 computer, introduced somewhat later.

IBM introduced its 7070 series in 1960. The 7070 represented a major step up from the first-generation 650 but it did not satisfy the very large number of 705 users. IBM eventually produced the 7080, a large transistorized machine that was compatible with the 705.

Second-generation technology made it possible to build small character-oriented processors at low cost. IBM's 1401, first delivered late in 1960, was the first of a very successful series of such computers. They started out as programmed controllers of input/output devices and developed into full data processing systems, especially when the more powerful 1410 and eventually the 7010 processors were introduced.

Other manufacturers followed with the introduction of small computers in numbers of models too numerous to discuss here.

TRANSITIONAL COMPUTERS. A number of second-generation computers were ahead of their time in introducing features usually associated with the third generation, even though they appeared early in the second generation. The Honeywell 800 introduced an ingenious hardware multiprogramming system along with a very interesting data processing software system, FACT.

The Atlas system, developed jointly by Ferranti and Manchester University in England, introduced the concept of virtual memory implemented through dynamic address translation.

The Burroughs 5000 system introduced a different implementation of virtual memory along with pushdown stacks and other features that help in compilation and in multiprogramming.

The Third Generation. Integrated circuits

and large-scale integration (LSI) are the most striking technological developments of the current third-generation of computers. However, some very important computers of the third generation make little or no use of integrated circuits. In this discussion, all computers introduced on or after April 7, 1964, will be considered to be third-generation computers, along with several computers introduced earlier whose technology and system design were sufficiently advanced to permit most of their installations to survive into the 1970s.

THE IBM 360 AND 370. April 7, 1964, is the date on which IBM announced its System 360. The 360 was designed to replace all earlier IBM computers, and it represented a very major departure from IBM's second-generation systems. The 360 came in a number of compatible models. The Model 75 at the top of the line used conventional hardware sequencing techniques to implement a large instruction set. The other models (30, 40, 50, 65) used microprogramming in a variety of read-only memory systems to provide the same instruction set on computers with a wide range of memory and circuit speeds. The use of microprogramming in read-only storage also made it possible for these 360 models to run programs written for second-generation IBM computers by emulation, i.e., by hardware-assisted simulation.

Many features of the 360 have become standards in large segments of the computer industry. Among these are the use of eight-bit bytes for the representation of characters and the use of nine-track tapes. Multiple-spindle disk pack systems, first the 2314 and more recently the 3330, were introduced on the 360 and have been adopted, sometimes with variations, on other systems.

The 360 was tremendously successful, and there are many thousands of installations. New models have been introduced at intervals since the initial announcement, including some like the models 20 and 44 that were not quite compatible with the standard 360.

The 360 Model 67 followed shortly after the original 360 announcement, in response to the demand from universities and research laboratories for a large-scale "time-sharing" system. The Model 67 provides dynamic address translation that permits the implementation of "virtual memory" operating systems.

In response to the delivery of the first Control Data 6600 in the fall of 1964, IBM announced a 90 series of a very large and very fast 360. Several of Model 91 and Model 95 were delivered before the series was withdrawn in favor of the Model 85. The

5 introduced the cache, or buffer memory, which provided an automatic multilevel memory system to help match memory speed to the very fast arithmetic need. The 85 was at the top of the 360 line only very briefly. It was superseded in 1971 by the more powerful 195, which provided a 360 system that was competitive with the CDC 7600 in the scientific computer field.

During 1970, IBM announced its 370 line, which represented a relatively modest step up from the 360. All 360 programs, with a few exceptions, run on the 370. An important feature of the 370 was held back until August 1972, at which time it was revealed that dynamic address translation already existed on the delivered models 135 and 145 and would be standard on new 370 models.

Two new large 370 systems, the 158 and 168, were introduced, whose major difference from the earlier 155 and 165 was the replacement of core memory by faster and much cheaper metal oxide semiconductor (MOS) memories. The MOS large-scale integration technology may very well lead to the rapid obsolescence of magnetic cores in computer memories.

Gene M. Amdahl, one of the designers of IBM's 360 series, left IBM in the fall of 1970 and started a new computer company, Amdahl Corporation. Major financial backing was obtained from Japanese sources. Amdahl's goal was to produce a computer compatible with the IBM 370 series, that would sell at roughly the same price as the Model 168, but would be considerably faster than the 168. The Amdahl computer uses existing IBM 370 software and peripherals. At least one Amdahl 470 V/6 system had been installed by the summer of 1975.

RCA AND UNIVAC. RCA's strategy in the third generation was to accept most features of the IBM 360 as standards for the industry, and to attempt to become an alternate source of supply for users who found that type of equipment attractive. The company introduced the Spectra 70 series whose principal models, the 35 and 45, were designed to fill, respectively, between the IBM 360/30 and 40 models and between the IBM 360/40 and 50. A virtual memory system, the Model 46, was also developed. RCA had only moderate success with the Spectra line, and a series of new virtual memory models introduced in advance of the IBM 370 series met with poor customer response. RCA abruptly departed from the computer business in the fall of 1971, and shortly thereafter Sperry-Rand's UNIVAC Division announced that it would purchase the remnants of RCA's computer division and would provide support for Spectra series installations.

UNIVAC's own entry into the third generation was with the 1108, a compatible extension of its second-generation 1107. The 1108 and its successor in the 1970s, the 1110 (Fig. 7), have made UNIVAC an important factor in the large-scale scientific computer field.

Another important UNIVAC series has been the 400, which has been used in large real-time and control applications.

In the very important small-to-medium scale data processing field, the UNIVAC third-generation 9000 systems followed closely the pattern set by IBM 360 systems. The new models in this line may prove attractive to those of the newly acquired RCA installations that choose to remain with UNIVAC.

CONTROL DATA CORPORATION. The CDC 6000 series easily qualifies as belonging to the third generation, even though the first 6600 was delivered in 1964 and even though the discrete component technology used is more typical of the second generation. For several years after its introduction the 6600 was faster and more powerful than any other computer available, and CDC established a strong position in the large scientific computer field. The speed of the 6600 (estimated by the manufacturer at three million instructions per second) was enhanced by the use of an instruction stack along with multiple arithmetic and logical units. The same system without these features was offered as a lower-priced 6400 system and a multiprocessor 6500 system. All these systems could use a very high speed extended core storage (ECS), a peripheral storage system with transfer rates up to ten million 60-bit words per second.

By 1969, Control Data had delivered its first 7600 system (Fig. 8). The 7000 series provides a good deal of compatibility with the 6000 series at three to seven times the speed of the 6600. Typical Atomic Energy Commission installations use the 6600 as a front-end computer for the faster 7600. Slight upgrades of these machines were marketed in the early 1970s as the CDC Cyber 70 Series. A faster compatible series of computers using integrated circuit technology was introduced as the Cyber 170 Series in 1974.

Another very large computer, the CDC STAR (String Array) 100 was built for Livermore and offered to other customers. The STAR is based on a concept of streaming arrays of data through pipeline arithmetic units at very high rates of speed, and it should prove very effective for some classes of problems. Texas Instruments has developed the Advanced Scientific Computer (ASC), whose size and speed are comparable to the STAR 100 for

DIGITAL COMPUTERS



Fig. 7. Console and CPU of UNIVAC 1110 system.

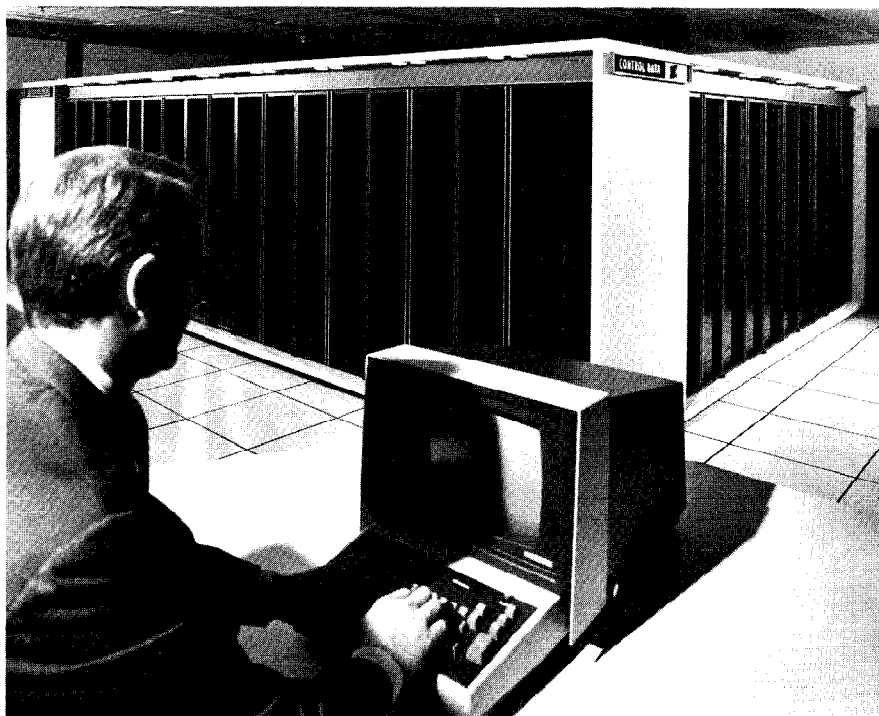
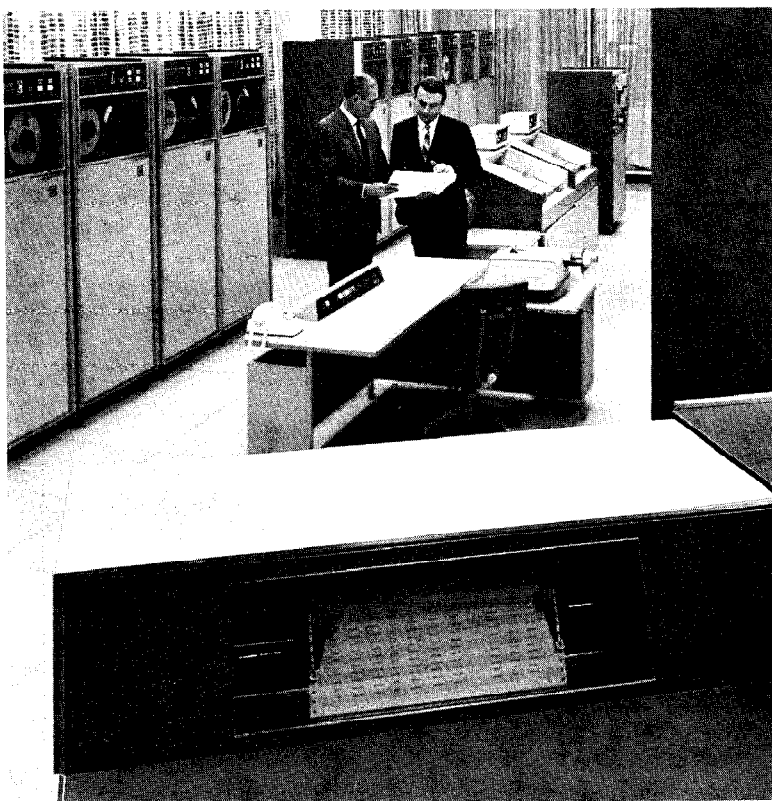


Fig. 8. CDC 7600 system.



(a)



(b)

Fig. 9. The Burroughs Corporation B5500(a) and B6700(b) systems.

DIGITAL COMPUTERS

similar classes of problems.

Control Data has also had reasonable success with the third-generation versions of its 3000 series computers in the small- and medium-size computer field.

Seymour Cray, who was the principal architect of the CDC 6600 and 7600 computers, left Control Data in 1972 to start a new company, Cray Research, Inc. The first product of the company, the Cray-I, is reported to be at least five times as fast as the 7600, with a price roughly the same as the 7600 had when it was first delivered. The first Cray-I was scheduled to be installed at Los Alamos late in 1975.

BURROUGHS. The Burroughs 5000 system was upgraded to the 5500 in 1962. Burroughs introduced a fixed-head disk for system residence and for use in its virtual memory system. Fixed-head disks became an IBM 370 component many years later. During the 1960s Burroughs used the slogan, "Burroughs dares to be different," and its stack organization and use of descriptors for memory addressing were indeed unique in the industry. The 6500 system, announced in 1965, was slow in delivery and in performance, and the very ambitious 8500 never reached completion, but the 6500 was soon replaced by a more capable 6700 and 7700 series. Fig. 9 shows the Burroughs B5500 and 86700 systems.

Meanwhile, Burroughs made great progress with smaller series of computers, the 2500 and 3500, later upgraded to 2700 and 3700. These and the even smaller 1700 series systems have made Burroughs a major factor in the third-generation computer field.

Burroughs also has built the ILLIAC IV, a parallel system based on the use of a large number of synchronized, high-speed arithmetic units. For appropriate problems, the ILLIAC IV is potentially orders of magnitude faster than other computers.

HONEYWELL AND GENERAL ELECTRIC. Honeywell entered the third generation with its 200 computer, which provided upward compatibility with the IBM 1400 series. The 200 grew into a whole series of computers that were very successful in the data processing field.

General Electric's third-generation entry was the 600 series, which provided limited upward compatibility with the IBM 7000 series. GE tried to make a spectacular entry into the large time-sharing computer field with the 645, a computer designed in cooperation with the M.I.T. Multics project, but the IBM 360 Model 67 took away most of that market. GE introduced a 200 and 400 series whose major success was in the time-sharing field. By 1970 it was clear that GE was not making much progress in the computer field, and it sold its computer division to

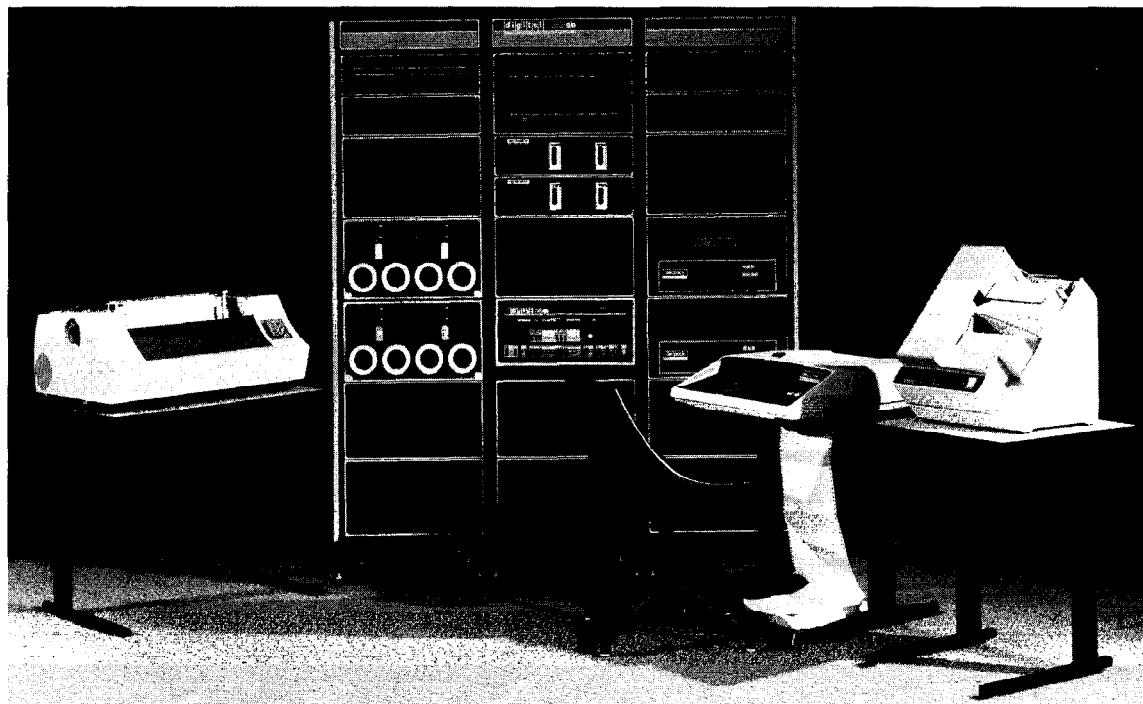


Fig. 10. DEC PDP-8/E minicomputer system.

Honeywell. The 600 series was upgraded to a 6000 series of more modern and more powerful computers, and this has helped Honeywell to become a leading contender in the computer industry.

MINICOMPUTERS. The most spectacular growth area in the latter part of the third-generation period has been in the minicomputer field. The largest company in this field is Digital Equipment Corporation, which is famous for its PDP series (Fig. 10). These started in the second generation, but achieved their major success with the third-generation PDP-8, which was first delivered in 1968 and which is installed in thousands of laboratories. The PDP-11 is a newer family of more powerful minicomputers introduced in 1970. These and other minicomputers have impinged on the medium-scale computer field, since large-scale integration technology is being applied to good effect to increase the power and speed of computers that are physically quite small. DEC also has had considerable success with its PDP-10, which is in the medium- to large-scale class. There are many other successful manufacturers of minicomputers, including Data General with their NOVA and Super-NOVA series, Hewlett-Packard, Varian, Interdata, and Microdata. Many minicomputer manufacturers use microprogramming techniques to increase the versatility of their products.

MICROCOMPUTERS. The development of large-scale integration has made it possible to develop quite elaborate computers on just one or two semiconductor chips. Small pocket calculators have been produced by Hewlett-Packard and others that contain the equivalent of tens of thousands of transistors. These calculators are the most conspicuous of the microcomputers that are being used in a very large number of applications.

Just about all areas of automatic control are potential users of microcomputers, whose cost/performance ratio is impressively low. Microcomputers make it possible to build low-cost intelligent terminals. They provide an alternative to many aspects of centralized time-shared computing. It is hard to estimate the full potential of the successors to the "computer on a chip" that first appeared in the early 1970s. Larger microcomputers may very well soon take over most functions of present-day minicomputers.

COMMUNICATIONS AND NETWORKS. An important characteristic of the third-generation computer is its adaptability to data communications. Remote entry of jobs and data has become almost routine. Computer service bureaus offer services of computer networks in national and international hookups.

Large corporations link their offices and dealers in elaborate data processing networks. Reservation systems and credit-checking systems have been constructed on a very large scale. Large data bases have been created or are planned with nationwide or even worldwide access by way of communication lines. Much of this type of communications traffic can be handled by standard telephone lines. Special data networks are being developed to provide for special high-speed, high-volume data transmission requirements. Computer network experiments carried on by ARPA and others in the 1960s and 1970s have provided prototypes for the very extensive, high-capacity data networks of the future.

Table 1 summarizes the three generations of computers discussed in this article.

Future Prospects. After many years of very rapid development and change in computers, it seems reasonably safe to predict a few years of consolidation, at least in the area of large computing systems, and especially those used in data processing.

There are a number of stabilizing factors in the computer industry. The large computer manufacturers have a very considerable investment in rented equipment. It is in the interest of the manufacturers, when they also function as leasing companies, to avoid too rapid obsolescence of existing equipment. Also, in spite of numerous antitrust suits, IBM's position in the industry continues to grow stronger, and IBM will be increasingly able to control the rate of technological change.

The very elaborate software systems that are now in use, and the increasing complexity of the software systems being developed, also have a stabilizing influence. A new and revolutionary computer, no matter how attractive, will not be considered at most large installations unless it can provide a full range of software products and a guarantee of relatively painless transition from the current system to the new one.

Because of the time lag inherent in the development and marketing of new large systems, it also seems reasonable to predict that the large computers to be used in 1980 will be the ones being marketed in the 1970s. Essentially they will be the same computers that are installed and running today.

There is a strong possibility that these very conservative projections may turn out to be wrong. The chief factor that may bring changes in the computer industry is the very vigorous small computer industry now developing. The major techno-

DIGITAL COMPUTERS

Table 1. Electronic computer generations

Development	Early First Generation. 1946–1953	Late First Generation, 1953-1959	Second Generation, 1959-1964	Early Third Generation, 1964-1969	Late Third Generation, 1969—
<i>Component technology</i>					
Vacuum tubes.	_____				
Transistors.			_____		
Hybrid circuits.				_____	
Monolithic integrated circuits.				_____	
Medium- and large-scale integration.					_____
<i>Main memory technology</i>					
Delay lines.	_____				
Electrostatic tubes.	_____				
Magnetic drums.	_____				
Magnetic cores.		_____	_____	_____	_____
Large-scale integration.					_____
<i>Main memory cycle time</i>					
4~0,000 μ s.	_____				
10–20 μ s.		_____			
2-10 μ s.			_____		
0.5-2 μ s.				_____	
0.020–1 μ s.					_____
<i>Peripheral storage</i>					
Magnetic tapes.	_____	_____	_____	_____	_____
Magnetic drums.	_____	_____	_____	_____	_____
Magnetic disks.		_____	_____	_____	_____
Laser and magnetic bubbles.					_____
<i>Software systems</i>					
Subroutine libraries,	_____	_____	_____	_____	_____
Interpreters.	_____	_____	_____	_____	_____
Assemblers.		_____	_____	_____	_____
Compilers.		_____	_____	_____	_____
Operating systems.		_____	_____	_____	_____
Multiprogramming and time-sharing communications systems (networks).			_____	_____	_____
<i>Special features</i>					
Interrupt systems.			_____	_____	_____
Virtual memory.				_____	_____
Microprogramming.				_____	_____
<i>Typical examples</i>					
ENIAC,EDVAC	IBM 650,704,	Philco 2000	Burroughs 5500	IBM 370 series,	
SEAC,SWAC	705,709	CDC 1604,3600	CDC 6000 series,	System 3	
Harvard Mark	UNIVAC II,	IBM 7000,1400	3300	CDC Cyber 70	
III,IV	1103A,SS80	series	IBM 360 series	series	
IAS machine	Burroughs 205,	Ferranti Atlas	UNIVAC 1108	DEC PDP- 10,11	
UNIVAC I, 1103	220	RCA 301,501	Honeywell 200	Honeywell 2000,	
Whirlwind	NCR 120,200	Honeywell 800	series	6000	
IBM 701,702	series	UNIVAC III,	RCA Spectra 70	UNIVAC 1110,	
	Datamatic 1 000	1107	NCR Century	9400	
	RCA Bizmac		G.E. 400,600	Burroughs 6700,	
	Many magnetic			1700	
	drum computers			Many	
				minicomputers	

ogical advance in the past few years has been the development of large-scale integration. For years engineers have been predicting revolutionary changes in the computer industry as a result of LSI, but the actual changes, though important, have been slow in coming. As noted above, it is now possible to produce relatively sophisticated computers on just a few semiconductor chips. Some of today's **mini-computers** are more powerful than many full-scale second-generation computers.

The manufacturers of the larger computing systems may be in for a shock if the smaller computers begin to compete with the larger ones in speed and capacity, though probably not in price. It is possible that the rapid pace of technological development in the small computer field will force a major upheaval in the whole industry, so that, rather than settling down, the industry will experience an even more rapid rate of change and growth than it has had in the past. A factor that may be significant is the impending entry of the Japanese electronics industry into the small- and medium-scale computer market.

Probably the most far-reaching changes in the computer field will be brought about by breakthroughs in speed, capacity, and price of peripheral storage. Large-scale disk storage systems still cost hundreds of thousands of dollars and provide access times in ten's of milliseconds to less than one billion characters. Recently developed laser technology promises access times in microseconds to larger memories at lower costs. Prototype laser memories are still in the experimental stage and there is little likelihood that they will soon supplant present systems. Regardless, it seems inevitable that the replacement of disks and drums by electronic storage will occur and will usher in a new computer generation.

S. ROSEN

ORIGINS

For articles on related subjects see **DIGITAL COMPUTERS**; Contemporary and Future, and Early; **MANUFACTURERS, COMPUTER**; and **STORED PROGRAM CONCEPT**.

For articles on related terms and biographical information see **AIKEN, HOWARD**; **BABBAGE, CHARLES**; **ECKERT, J. PRESPER**; **EDVAC**; **ENIAC**; **HOLLERITH, HERMAN**;

LEIBNIZ, GOTTFRIED WILHELM; **MARK I**; **MAUCHLY, JOHN W.**; **PASCAL, BLAISE**; **TURING, ALAN**; **VON NEUMANN, JOHN**; and **ZUSE, KONRAD**.

Mechanical aids to calculation and mechanical sequence-control devices were perhaps the earliest and most important achievements in the development of computer technology.

The first adding machines date from the early seventeenth century, the most famous of which was invented by the French scientist and philosopher Blaise Pascal, although it is now believed that his work was predated by that of William Schickard. A number of Pascal's machines, which he started to build in 1642, still exist. Even though he had intended them for practical use, their unreliability caused them to be treated mainly as objects of scientific curiosity. During the subsequent two centuries, numerous attempts to develop practical calculating machines were made by Morland, Leibniz, Mahon, Hahn, and Müller, among others. However, it was not until the mid-nineteenth century that a commercially successful machine was produced. This was the "arithmometer" of Thomas de Colmar, the first version of which was invented in 1820, and which used the stepped-wheel mechanism invented by Leibniz.

Mechanical devices for controlling the sequencing of a set of operations, such as the rotating pegged cylinders still seen in music boxes today, date back even earlier. For example, de Caus (1576-1626) used such a mechanism to control both the playing of an organ and the movements of model figures. One of the most famous designers of mechanical automata was Vaucanson. In 1736 he successfully demonstrated an automaton that simulated human lip and finger movements with sufficient accuracy to play a flute. Vaucanson was also involved in the development of what came to be known as the Jacquard loom, in which the woven pattern was specified and controlled by a sequence of perforated cards. The original idea can be traced back to Bouchon in 1725, but such automatic looms did not come into widespread use until early in the nineteenth century after the work by Jacquard.

In 1834 these two lines of development came together in the work of Charles Babbage, who had become dissatisfied with the accuracy of printed mathematical tables. Earlier, in 1822, Babbage had built a small machine, involving several linked adding mechanisms, which would automatically generate successive values of simple algebraic functions, using the method of finite differences. His attempt at

DIGITAL COMPUTERS

making a full-scale model with a printing mechanism was abandoned in 1834, and he then started to design a more versatile machine. In the space of a few years he had developed the concept of a program-controlled, mechanical, digital computer, incorporating a complete arithmetic unit, store, punched-card input and output, and printing mechanism. The machine, which he called an analytical engine, was to have been controlled by programs represented by sets of Jacquard cards, with conditional jumps and iteration loops being provided for by devices that skipped forward or backward over the required number of cards. Internally, the machine was essentially microprogrammed by rotating pegged cylinders that controlled the sequencing of subsidiary mechanisms.

Babbage's work inspired several other people, among whom were Ludgate, who designed an analytical engine in Ireland in 1909; Torres y Quevedo, who, demonstrated the feasibility of an electromechanical analytical engine by successfully producing a typewriter-controlled calculating machine in 1920; and Couffignal, who started to design a binary analytical engine in France during the 1930s. However, Babbage's pioneering efforts were apparently unknown to most of the people who worked on the various computer projects during World War II and who were unaware that the problems they were tackling had been considered and often solved by Babbage more than a hundred years earlier.

The Jacquard loom was perhaps the source of Herman Hollerith's idea of using punched cards to represent logical and numerical data, developed for use in the 1890 U.S. National Census. His system, incorporating hand-operated tabulating machines and sorters, was highly successful and spread rapidly to several other countries. Automatic card-feed mechanisms were soon provided, and the system began to be used for business accounting applications. Following a dispute with Hollerith, the Bureau of the Census developed in time for the 1910 Census a new tabulating system involving mechanical sensing of card perforations, as opposed to Hollerith's system of electrical sensing. James Powers, the engineer in charge of this work, eventually left the Bureau to form his own company, which eventually became part of Remington Rand. Hollerith's company merged with two others to become the Computing-Tabulating-Recording Company, which in 1924 changed its name to the International Business Machines Corporation.

In 1937 Howard Aiken of Harvard University approached IBM with a proposal for a large-scale calculator, to be built from the mechanical and

electromechanical devices that were used for punched-card machines. The resulting machine, the Automatic Sequence Controlled Calculator, or Harvard Mark I, was built at the IBM Development Laboratories at Endicott. The machine, which was completed in 1943, was a huge affair with 72 decimal accumulators, capable of multiplying two 23-digit numbers in 6 sec. It was controlled by a sequence of instructions specified by a perforated paper tape; somewhat surprisingly, in view of Aiken's knowledge of and respect for Babbage's efforts, it lacked general conditional jump facilities. After completion of the Mark I, Aiken and IBM pursued separate paths. Several more machines were designed at Harvard, the first being another tape-controlled calculator, built this time from electromagnetic relays. IBM produced various machines, including several plug-board-controlled relay calculators and the partly electronic Selective Sequence Electronic Calculator, which was very much in the tradition of the original Mark I.

Not until well after World War II was it found that in Germany there had been an operational program-controlled calculator built earlier than the Mark I, namely, Konrad Zuse's 23 machine, which first worked in 1941. This machine, which had been preceded by two earlier but unsuccessful machines, had a mechanical store, but was otherwise built from telephone relays. It could store 64 floating-point binary numbers, and has been described as somewhat faster than the Harvard Mark I. The **Z3**, like several other machines built by Zuse, did not survive the war; the only one of Zuse's machines to do so was the **Z4** computer, which was later used successfully for several years at the Technische Hochschule in Zurich.

Various other electromechanical machines were built during and even after World War II, including an important series of relay calculators at the Bell Telephone Laboratories. The first of these, the Complex Computer, was demonstrated in September 1940 by being operated in its New York City location from a teletypewriter installed in Hanover, New Hampshire, on the occasion of a meeting of the American Mathematical Society. The Complex Computer, or Model 1, was capable of adding, subtracting, multiplying, and dividing two complex numbers, but lacked any sequence-control facilities. Later machines in the series incorporated successively more extensive sequencing facilities, so that the Model 5 relay calculator was a truly **general-purpose** (tape controlled) computer that achieved very high reliability of operation.

The earliest known electronic digital calculating

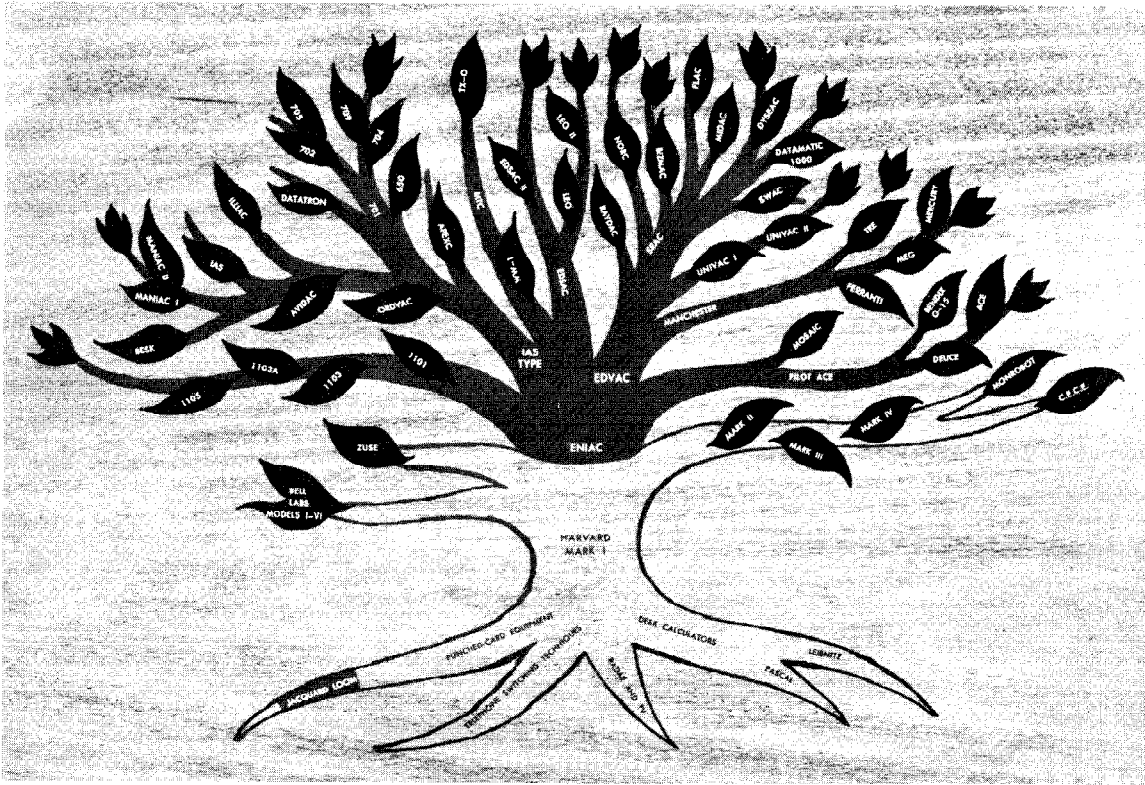


Fig. 1. Family tree of computers to mid-1950s. (Courtesy of the Smithsonian Institution.)

device was a machine for solving up to 30 simultaneous linear equations, initiated in 1938 at Iowa State College by John Atanasoff and Clifford Berry. Although the arithmetic unit had been successfully tested before the project was abandoned in 1942, the input/output mechanism was still incomplete, so the machine never saw actual use. Other important work on the development of electronic calculating devices was done at IBM, starting in 1942 with the building of experimental versions of various punched-card machines, including a multiplier. This machine was the origin of the electronic multipliers and calculating machines, such as the Type 604 and the Card Programmed Calculator (CPC), which IBM produced in great quantities in the years immediately following World War II and which played an important role until stored program electronic computers became widely available.

The earliest known efforts at applying electronics to a general-purpose, program-controlled computer were those undertaken by Schreyer and Zuse in 1939, but their plans for a 1,500 valve machine were later rejected by the German government. In Britain, a series of large special-purpose

electronic computers, intended for code-breaking purposes, was developed by a team at Bletchley Park, with which Alan Turing was associated. The first of these machines, which incorporated about 2000 tubes, was operating in December 1943. It has been described as being, in a very limited fashion, a program-controlled device. Interestingly enough, several postwar British electronic computers were developed by people who had been involved with these secret machines.

However, by far the most influential line of development was that carried out at the Moore School of **Electrical Engineering** at the University of Pennsylvania by John Mauchly, J. Presper Eckert, and their colleagues, starting in 1943. This work, which derived at least as directly from Vannevar Bush's prewar mechanical differential analyzer as from any digital calculating device, first led to the development of the **ENIAC**, which was officially inaugurated in February 1946. This machine was intended primarily for ballistics calculations, but by the time it was completed, it was really a **general-purpose** device, programmed by means of pluggable interconnections. Its internal electronic memory

DIGITAL COMPUTERS

consisted of 20 accumulators, each of 10 decimal digits, and it could perform 5,000 arithmetic operations per second—it was approximately a thousand times faster than the Harvard Mark I. The ENIAC was very much the most complex piece of electronic equipment that had ever been assembled, incorporating 19,000 tubes, and using nearly 200 kW of power. The machine was very successful, despite earlier fears regarding the reliability of electronic components.

However, even before the ENIAC was complete, the designers, who had been joined by John von Neumann, started to plan a radically different successor machine, the EDVAC. The EDVAC was a serial binary machine, far more economical on electronic tubes than ENIAC, which was a decimal machine in which each decimal digit was represented by a ring of ten flip-flops. A second major difference was that EDVAC was to have a very much larger internal memory than ENIAC, based on mercury delay lines. For these reasons, the initial design of EDVAC included only one-tenth of the equipment used in ENIAC, yet provided a hundred times the internal memory capacity.

It was apparently the discussions of the various ways in which the capabilities of ENIAC might be extended, together with the knowledge of the possibility of comparatively large internal memories, that led to realization that sequence-control information could be represented by words held in memory along with the numerical quantities entering into the computation, rather than by some external means such as perforated tape or pluggable interconnections. Thus, EDVAC could retain the great speed of operation that had been achieved by ENIAC, but could avoid the very lengthy setup time, often on the order of a day or more, that had made it impractical to use for other than very extensive calculations. The fact that a program could read and modify portions of itself was heavily utilized, since ideas such as index registers and indirect addresses were still in the offing. Of more lasting significance was the practical and attractive proposition of using the computer to assist with the preparation of its own programs.

With EDVAC, therefore, the invention of the modern digital computer was basically complete. The plans for its design were widely published and extremely influential, so that even though it was not the first stored-program electronic digital computer to be put into operation, it undoubtedly was the major initial inspiration that started the vast number of computer projects during the late 1940s.

REFERENCES

1961. Morrison, P., and E. Morrison (Eds.). *Charles Babbage and His Calculating Engines: Selected Writings* by Charles Babbage and Others. New York: Dover.
1968. de Beauclair, W. *Rechnen mit Maschinen: Eine Bildgeschichte der Rechentechnik*. Vieweg, Braunschweig.
1969. Rosenberg, J. M. *The Computer Prophets*. New York: Macmillan.
A popular account of the work of many of the computer pioneers.
1972. Goldstine, H. H. *The Computer from Pascal to von Neumann*. Princeton: Princeton University Press.
1973. Randell, B. (Ed.). *The Origins of Digital Computers*. Berlin: Springer.

B. RANDELL

EARLY

For articles on related subjects see **DIGITAL COMPUTERS**, Contemporary and Future, and Early; **MANUFACTURERS, COMPUTER**; and **STORED PROGRAM CONCEPT**.

For articles on related terms see **AIKEN**, **HOWARD**; **ECKERT, J. PRESER**; **EDSAC**; **EDVAC**; **ENIAC**; **MARK I**; **MAUCHLY**, **JOHN W.**; **READ-ONLY STORE**; **REGISTER**; **TURING, ALAN**; **ULTRASONIC MEMORY**; **UNIVAC I**; **VON NEUMANN, JOHN**; **WHIRLWIND**; and **ZUSE, KONRAD**.

The digital computer age began when the Automatic Sequence Controlled Calculator (Harvard Mark I) started working in August 1944. This machine was based on the mechanical technology of rotating shafts, electromagnetic clutches, and counter wheels, developed over the years for punched card tabulating machinery. It was constructed by IBM, following the ideas of Howard Aiken, whose original proposals go back at least to 1937. The shaft rotation period, and hence the time required to transfer a number or perform an addition, was 0.3 sec, while multiplication and division took 6 and 11.4 sec, respectively.

No other large machines using rotating shafts were built, but there were a number of successful magnetic relay machines. Bell Telephone Labora-

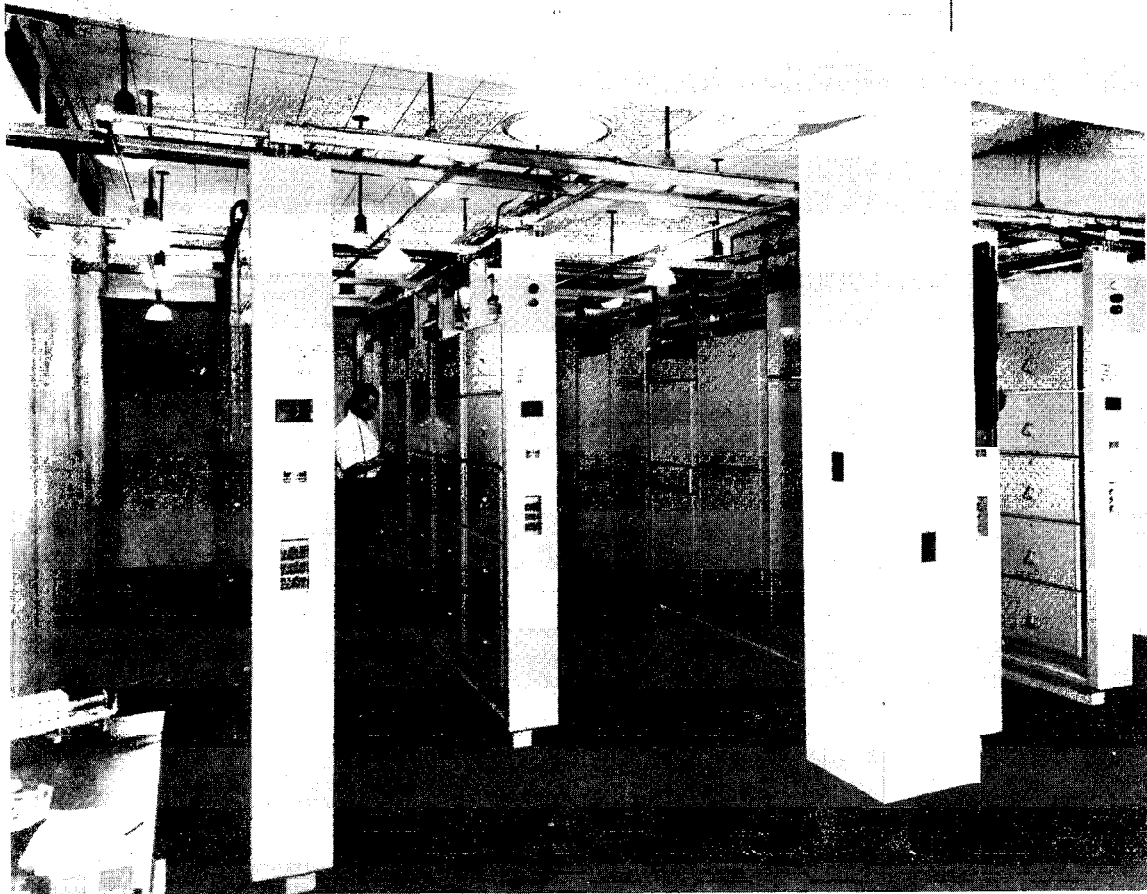


Fig. 1. The second Bell Model V relay calculator installed at Aberdeen Proving Ground. The first was installed at Langley Field, Virginia.

tories had been working in this area since 1938. Their first fully automatic computer was the one now referred to as the Bell Model V (Fig. 1), of which two examples were constructed. The first of these began to work at the end of 1946. An addition took 0.3 sec and multiplication and division took up to 1.0 and 2.2 sec, respectively. The last of the series was the Model VI, commissioned in 1949. Harvard Mark II, a relay machine designed by Aiken and following a very different design philosophy, was running in September 1948. A relay computer constructed in Sweden (BARK) was operational early in 1950. Independent work on relay computers had also been done by K. Zuse in Germany, and a Zuse 24 was running in Zurich in 1950. Relays lend themselves to complex circuit arrangements, and all the machines just mentioned had floating-point arithmetic operation, a feature that did not appear in electronic computers until well after the period now

under review here. The Bell machines had elaborate checking arrangements, including a redundant representation for stored numbers. Model VI even had a re-try feature, designed to mitigate the effect of transient relay faults.

The concept of the large-scale electronic computer is due to J. Presper Eckert and John W. Mauchly. They were already building the ENIAC when the Harvard Mark I was commissioned. The ENIAC contained nearly 19,000 vacuum tubes, more than twice as many as any later vacuum-tube computer. Because it was by far the most complex machine constructed up to that time, its construction was a great act of technological courage, both on the part of the designers and of the Moore School of Electrical Engineering in Philadelphia where it was constructed. The ENIAC began to function in the summer of 1945. An addition took 200 μ s and a multiplication took 2.8 ms.

DIGITAL COMPUTERS

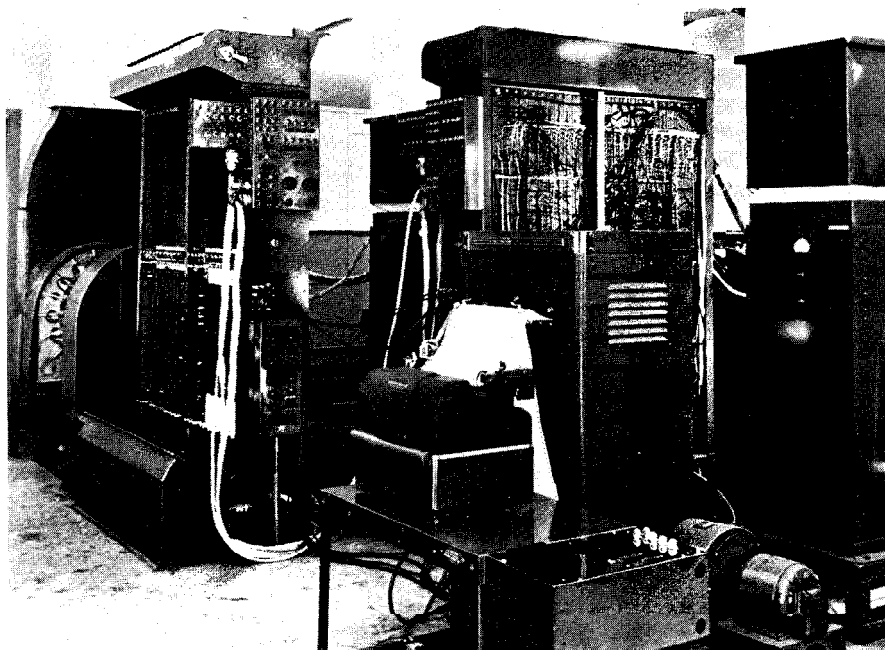


Fig. 2. The BINAC computer.

Table 1. Characteristics of electronic computers (as of early 1951)

Computer	Serial or Parallel	Decimal or Binary	No. of Addresses	Word length	Clock frequency KH	Memory	
						Type	No. of Words
EDVAC ^(b)	S	B	3 + 1 ^(d)	44 bits	1,000	U	1,024
UNIVAC	S	D	1	12 char.	2,250	U	1,000
IAS ^(b)	P	B	1	40 bits	Asynch.	W	1,024
EDSAC	S	B		35 bits	500	U	512
Ferranti I	S	B	1	40 bits	100	W	256
Pilot ACE	S	B	1 ^(d)	32 bits	1,000	U	360
SEAC	S	B	3	45 bits	1,000	U	512
SWAC	P	B	4	36 bits	125	W	256
Whirlwind I	P	B	1	16 bits	1,000	E	256
Harvard Mark III	S/P	D	3	16 dec.	28	D	4,000 ^(c)
Burroughs	S	D	1 or 1 + 1 ^(d)	9 dec.	125	D	800
ERA 1101	P	B	1 + 1 ^(d)	24 bits	400	D	16,384

Notes: (a) U = ultrasonic delay (mercury tank); W = Williams tube; D = magnetic drum; E = electrostatic (CRT)

(b) Not commissioned until 1952.

(c) Separate 200-word memory for instructions.

(d) Provision for minimum-access coding.

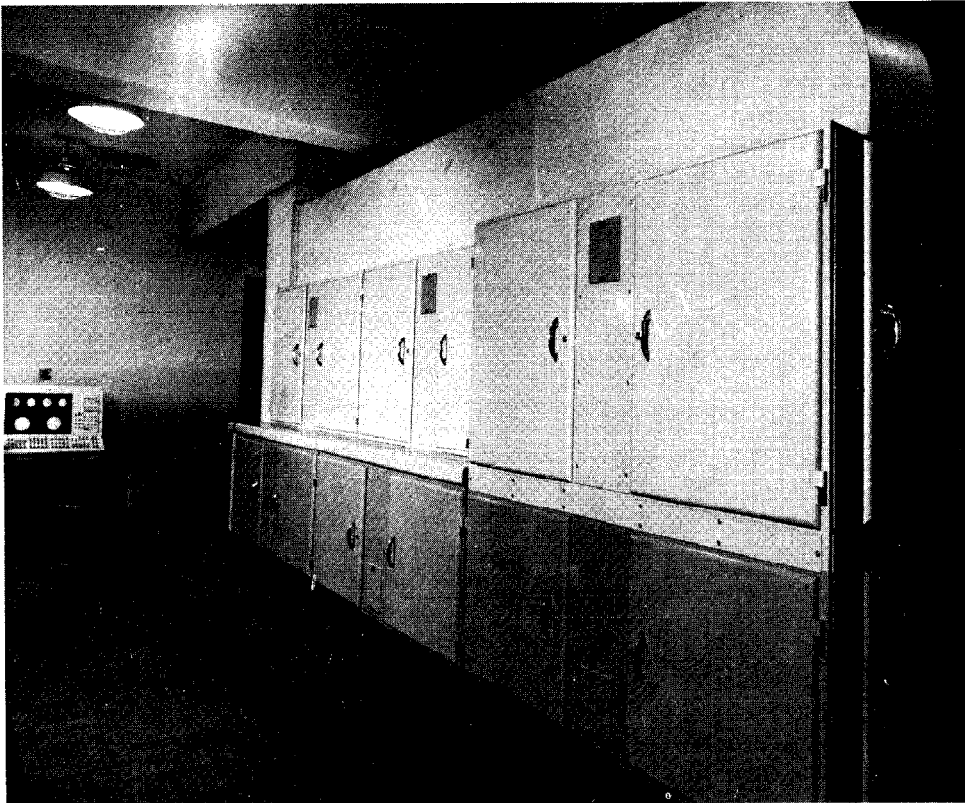


Fig. 3. The Ferranti Mark I computer at Manchester University, 1951.

Max. Memory Access, (time ms)	Operation Time (incl. access)			Input Output	No. of Tubes	No. of Diodes (germanium)	Aux. Memory
	Add, ms	Mult., ms	Divide ms				
0.38	0.2-1.5	2.2-3.5	2.2-3.6	Paper tape	3,600	10,000	-
0.40	0.5 mean	2.15 mean	3.9 mean	Magn. tape	5,600	18,000	Magn. tape
0.025	0.062	0.44-1.0	1.1	Cards	2,300	0	-
1.1	1.5 mean	6 mean	-	Paper tape	3,800	0	-
0.64	1.2	3.36	-	Paper tape	3,800	0	Drum, 16K
1.0	-	2	-	Cards	800	-	-
0.38	1.5 max.	3.6 max.	3.6 max.	Paper tape	1,300	15,800	Magn. tape
	0.064	0.38	-	Paper tape; cards	2,300	3,000	
0.016	0.049	0.06-1	0.1	Paper tape	6,800	22,000	
4.5	5	13	100	Magn. tape	5,000	1,300	-
32	0.6-17	30-50	-	Paper tape	3,271	6,773	
17	0.1 min.	0.35 min.	0.42 min.	Paper tape	2,200	3,000	

DIGITAL COMPUTERS

The very early computers were extremely limited in the amount of internal storage that they had. Provision was usually made for tables to be held in read-only storage (banks of switches or punched paper tape) with arrangements for interpolation. It was frequently possible for the programmer to arrange that more than one arithmetic or transfer operation should take place at the same time. The ENIAC was programmed by setting up hundreds of plugs and sockets and switches, an operation that could take several hours. The other computers read their instructions from punched paper tape, endless loops being used for repeated sections of the program.

While the ENIAC was still under construction, Eckert and Mauchly began to realize that, by the application of logical principles, it would be possible to construct a machine not only much more powerful than the ENIAC but also much smaller. They were joined by John von Neumann on a part-time basis, and it was from the group so formed that the ideas of the modern stored-program computer emerged. They were summarized in a document entitled "First draft of a report on the EDVAC," prepared by von Neumann and dated June 30, 1945.

Eckert and Mauchly did not stay at the Moore School to work on the EDVAC, and it was not until January 1952 that a machine bearing that name was commissioned. Instead, they founded the Eckert-Mauchly Corporation, with the object of designing and marketing the UNIVAC. This company was finally absorbed into Remington Rand, but the name UNIVAC has happily survived.

From the beginning the UNIVAC was designed with an eye to business data processing, and the standards set for performance and reliability were very high. In March 1951, the first UNIVAC passed a rigorous acceptance test and was delivered to the U.S. Bureau of Census. It was then a fully engineered machine, with magnetic tape and other peripherals required for large-scale business operations. The Eckert-Mauchly Corporation had demonstrated a smaller machine, the BINAC (Fig. 2), in August 1949, but this was not very successful and they decided to concentrate their efforts on the UNIVAC.

When the Moore School group broke up, von Neumann established at the Institute for Advanced Study, Princeton, a project for the construction of a computer. Von Neumann himself, assisted by H. H. Goldstine, laid down the logical structure of this computer, and the engineering development and design was in the hands of J. H. Bigelow. It was the first parallel computer to be designed, and it intro-

duced techniques that are now commonplace, such as the register economizing device of putting the multiplier in the tail of the accumulator and shifting it out as the multiplication proceeds. Although the machine was not working until October 1952, the project had immense influence on the development of the digital computer field. The ultrasonic memory, which had been proposed for the EDVAC, was thought to be too slow for a parallel machine, and it was planned to use instead a memory based on the Selectron proposed by J. A. Rajchman. The Selectron did not fulfill its promise, but fortunately the Williams tube memory came along in time to save the situation.

The experimental computers that came into action first were those that were least ambitious, both in specification and in performance. One of these was the EDSAC, a computer directly inspired by the EDVAC, designed and constructed by myself and W. Renwick in Cambridge, England. This computer did its first calculation on May 6, 1949, and was used for much early work on the development of programming techniques. Activity at Manchester University arose out of work by F. C. Williams on what became known as the Williams tube memory. In order to test this system, Williams and T. Kilburn built a small model computer with a memory of 32 words and with 5 instructions in its instruction set. The only arithmetic instruction was for subtraction. Development work continued, and by the summer of 1949 a computer with a magnetic drum as a backing memory was demonstrated. The Ferranti Mark I computer (Fig. 3), of which the first delivered model was inaugurated at Manchester University in July 1951, was based on this work.

A third center of activity in England was at the National Physical Laboratory, where the inspiration came from Alan Turing. Turing did not stay there long, leaving for Manchester University in 1948, but the Pilot ACE, which was running by December 1950, reflected very strongly his rather personal view of computer design. The Pilot ACE used an ultrasonic memory, and it was necessary for the programmer to know more of the structure of the machine and the timing of pulses within it than was required in the case of other machines.

The first of the American machines to be brought into use was the SEAC, dedicated on June 20, 1950. This was built under the direction of S. N. Alexander at the National Bureau of Standards in Washington and the success of that group is the more remarkable since the SEAC project started after many others. The SEAC was elegant in design and construction, and pioneered the use of small

plug-in packages; each package contained a number of germanium diodes and a single vacuum tube. The SEAC used an ultrasonic memory, but a Williams tube memory was later added for evaluation purposes. Meanwhile, H. D. Huskey, who had formerly been a member of the team at the National Physical Laboratory in England and had worked on ENIAC, was completing the SWAC at the NBS Institute for Numerical Analysis at UCLA. This was a parallel machine with a Williams tube memory and was very fast by the standards of the day.

Whirlwind I was a computer with a short word length, aiming at very high speed and power, and intended ultimately for air traffic control and similar applications. It was designed and built under the direction of J. W. Forrester at M.I.T. and was operating in December 1950. From its specification, one would take it to be the first of the mini-computers, but in fact it occupied the largest floor area of all the early computers, including the ENIAC. The memory was of the electrostatic type, but the cathode-ray tubes were of special design and operated on a different principle from that used by Williams.

Table 1 gives brief particulars of the computers mentioned above and also of several additional ones that became operational in the same period.

REFERENCES

1951. U.S. Navy, Office of Naval Research. *Digital Computer Newsletter*, vols. 1-3.
1953. U.S. Navy, Office of Naval Research. *A Survey of Automatic Digital Computers*.
1972. Goldstine, H. H. *The Computer from Pascal to von Neumann*. Princeton: Princeton University Press.

M. V. WILKES

DIGITAL-TO-ANALOG CONVERTERS

For articles on related subjects see ANALOG COMPUTERS; DATA COMMUNICATIONS; DIGITAL COMPUTERS; and HYBRID COMPUTERS. For article on related term see BINARY CODED DECIMAL, NATURAL.

Whenever it is necessary to communicate between analog and digital systems, analog-to-digital

(A-D) and/or digital-to-analog (D-A) converters are required. These converters form basic links between the world of "real" phenomena, where the variables are generally continuous analog quantities, and the "engineer designed" world of digital information processing and data communications, where the variables are discrete quantized quantities,

The number of applications and types of converters available has grown significantly during the past few years. In part, this has resulted from increased recognition of the capabilities of digital, as opposed to analog, signal processing and data transmission. The importance of these capabilities is application dependent; however, in general, the advantages of digital processing and transmission lie in the increased accuracy, noise immunity, processing flexibility, and storage facilities afforded by the digital format. This increasing use of digital processing of analog signals has been aided by the rapid development of sophisticated yet inexpensive minicomputers. At the same time the steady decline in price and increase in the performance of A-D and D-A converters has allowed minicomputers to be effectively coupled to the analog world.

Some Applications. Successful and widespread use of digital processing has resulted in numerous examples of A-D and D-A converter use. A simple classification of application areas is given below (also see Hoeschele, 1968).

DIGITAL CONTROL SYSTEMS. Fig. 1 is a block diagram illustration of a digital control system. Variables originate within the plant or system. They are sensed by an analog sensor, digitized by an A-D converter, and then transmitted to a digital processor. If the processor merely manipulates and stores this information, then the system is a simple data acquisition system. If, on the basis of the input information, control signals determined by the processor are returned to the plant, then a digital control system is present. A variation on this system, which requires fewer converters, can be designed if the signal frequencies and number of sensors and controllers are not excessive (Fig. 2). Such control systems can be found in a wide variety of situations, from basic industrial processing to aerospace flight systems.

HYBRID COMPUTATION SYSTEMS. Hybrid computers consist of an analog computer and a digital computer communicating to each other through a fairly sophisticated interface. This interface normally includes several A-D and D-A converters for transforming the signals to the appropriate computer format. While the analog computer is a low-accuracy

DIGITAL-TO-ANALOG CONVERTERS

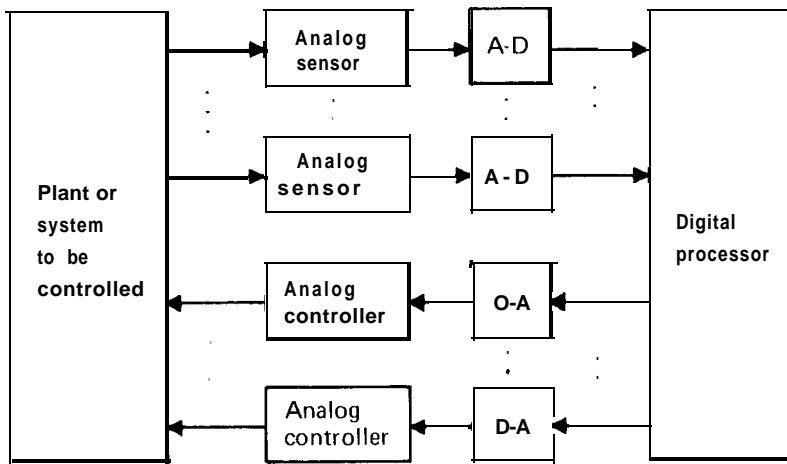


Fig. 1. Digital control system.

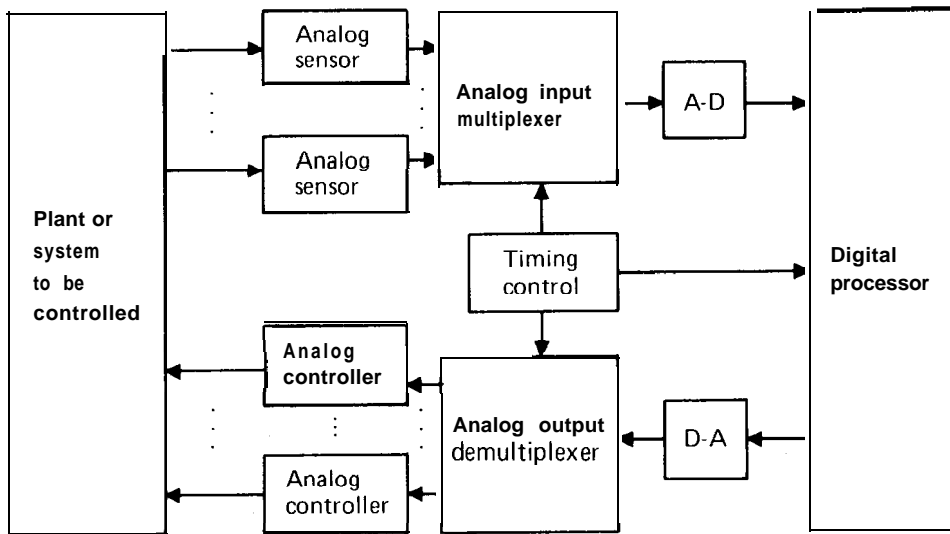


Fig. 2. Digital control system with multiplexers.

device, it does permit fast parallel solution of ordinary differential equations. The digital computer, on the other hand, is a high-accuracy serial machine with extensive logic and memory capabilities. Together, communicating through A-D and D-A converters, they permit very efficient solution of certain classes of continuous system optimization and statistical problems. Converters used in this application are often designed with computational capabilities. Thus, a D-A converter may act as a multiplier in addition to a converter.

COMMUNICATIONS SYSTEMS. The advantages of digital data transmission have resulted in extensive

use of converters as parts of telemetering and voice communications systems. In telemetering systems, analog signals originating in remote locations are first converted into digital signals and then transmitted to the control station. Remote weather and defense-related monitoring systems fall in this category of applications.

Voice communications systems are also becoming increasingly oriented toward digital signal processing. Thus, in many situations, analog voice signals are being digitized with A-D converters and subsequently transmitted over time-shared channels, with many conversations being "simultaneously"

carried over the same channel. Such systems can be designed to be flexible, and can handle both speech and data at the same time while making "optimum" use of the systems bandwidth capabilities.

TEST, MEASUREMENT, AND MONITORING. In contrast to monitoring systems that require extensive communications capabilities, many applications of A-D converters can be found in test and measurement equipment. Digital voltmeters, for example, have gained widespread acceptance during the past few years. More complex measurement and monitoring applications such as on-line, real-time patient monitoring also have converters as key system elements.

The Basic Relationship. Analog variables such as position, temperature, and process rate are typically first converted during measurement into analog voltages and currents. Conversely, to control the analog variables, analog voltages and currents are usually supplied to the inputs of a controlling transducer. Rather than deal with the basic analog variable (e.g., temperature), it is therefore convenient to deal with the voltages or currents available at the output, or produced for the input, of the transducer. The analog variable considered here is thus a pure voltage or current, and questions concerning transducer operation, signal amplification, and signal conditioning are omitted. Material on these important practical matters can be obtained from the references.

Digital information is generally represented by the presence or absence of a fixed voltage or current level. Thus, each unit of information, or "bit," has two states, referred to as the "one" and "zero" states. On a single input line, information can therefore be represented serially by periodically changing the voltage level or state of the line. A set of parallel lines or a grouping of serial bits can be used to represent a digital word where the meaning of this word depends on the number or symbol assigned to each possible combination of bits. This is referred to as the "code."

Different types of codes are used with A-D and D-A converters. However, for simplicity, this article considers only *natural binary* code. Table 1 presents this code for a 3-bit word. In general, each word may have n bits, with the leftmost bit-the most significant bit (MSB)-having a weight of 2^{-1} , the rightmost bit-the least significant bit (LSB)-having a weight of 2^{-n} , and the i th bit ($1 < i \leq n$) having a weight of 2^{-i} . Signed numbers maybe represented by adding an extra bit whose presence or absence indicates whether the number is negative or positive.

Table 1. Three-bit natural binary code

Decimal Value	Binary Value	BIT 1 (MSB)	BIT 2	BIT 3 (LSB)
0	.000	0	0	0
1/8	.001	0	0	1
2/8	.010	0	1	0
3/8	.011	0	1	1
4/8	.100	1	0	0
5/8	.101	1	0	1
6/8	.110	1	1	0
7/8	.111	1	1	1

The basic conversion relationship for a 3-bit binary code is given in Figs. 3(a) and 3(b). Thus, if the input to an A-D converter is properly scaled, then corresponding to any input is a distinct 3-bit code output.

Similarly, any 3-bit digital sequence entering into the D-A converter defined by Fig. 1 results in producing one of eight distinct voltage outputs. The *ideal resolution* of this converter is equal to the value of the LSB, or 2^{-n} , for an n -bit converter. Associated with this resolution is an inherent *quantization error* which reflects an uncertainty in the results of A-D conversion due to quantification of the analog signal. For the system of Fig. 3(a) this uncertainty is 1 LSB; however, if the zero position is offset $1/2$ LSB so that transitions occur in the middle of each voltage range, as shown in Fig. 3(b), then the quantization *error* becomes an optimum $\pm 1/2$ LSB. This corresponds to a rounding as opposed to a truncating operation.

In both D-A and A-D converters there is a wide variety of techniques and manufacturers. Davis (1972) provides a list of over 40 firms producing devices in this area. The following two sections discuss several of the more basic techniques used in the conversion process. More elaborate and expensive techniques must be used if very high speed or accuracy is desired.

D-A Converters. Fig. 4 shows a block diagram for a D-A converter. Every D-A converter contains switches and a resistor network. The switches are controlled by the digital input code, and establish connections within the network needed to obtain the proper analog voltage.

Fig. 5(a) shows a simple 3-bit plus sign D-A converter. The dashed lines indicate that the switch is controlled by the associated digital bit input. The switches themselves are generally integrated circuits, which ideally would have no resistance when closed and infinite resistance when open. For the 0100

DIGITAL-TO-ANALOG CONVERTERS

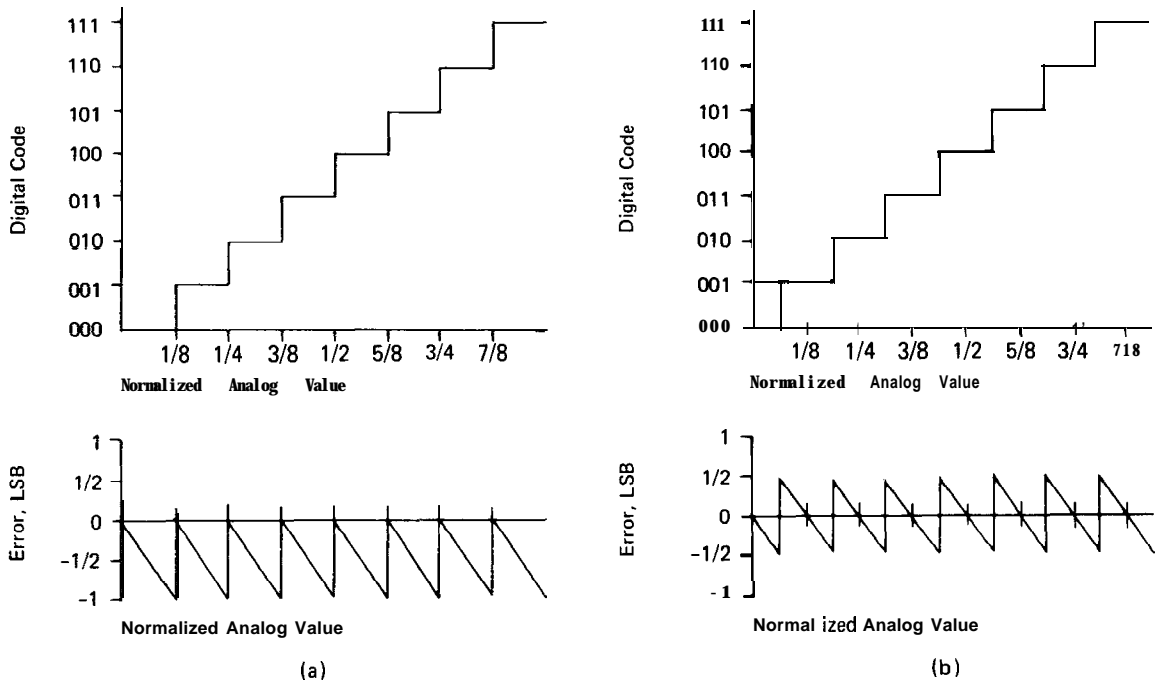


Fig. 3. The basic relationship. (a) Maximum quantization error 1 LSB. (b) Maximum quantization error $\frac{1}{2}$ LSB.

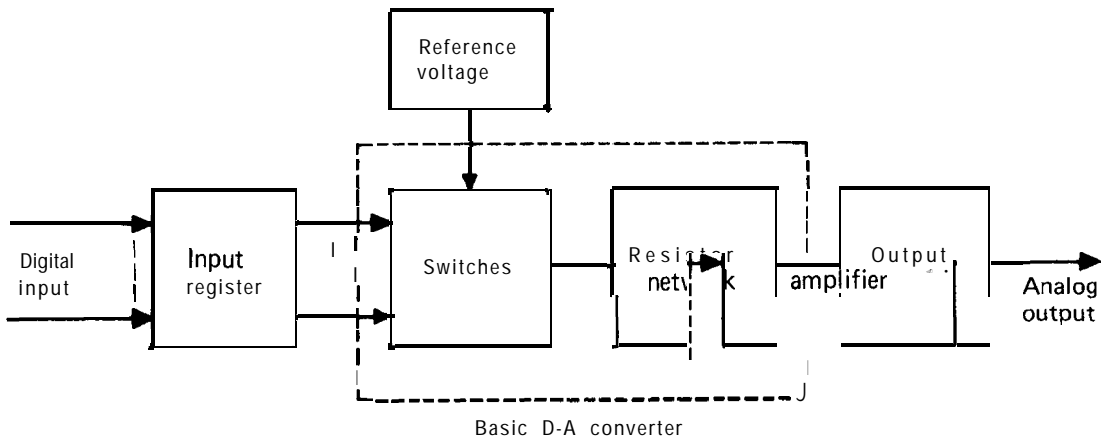


Fig. 4. Basic D-A converter and accessories.

input switch configuration shown, the output voltage V_o is easily seen to be $V_R/2$. Similarly the n th bit present can be shown to produce an output voltage increment equal to $2^{-n}V_R$; hence, the resulting output voltage is proportional to the binary input. The sign bit controls a voltage reference switch. With certain codes its absence indicates a positive digital input and results in switching in the positive reference

voltage $+V_R$. Its presence indicates a negative input and the negative reference voltage $-V_R$ is applied to the network.

Another simple D-A converter based on summing currents is shown in Fig. 5(b). This has the advantage of requiring only one resistor per bit; however, a large range of resistance values is necessary, making it less suitable to monolithic and

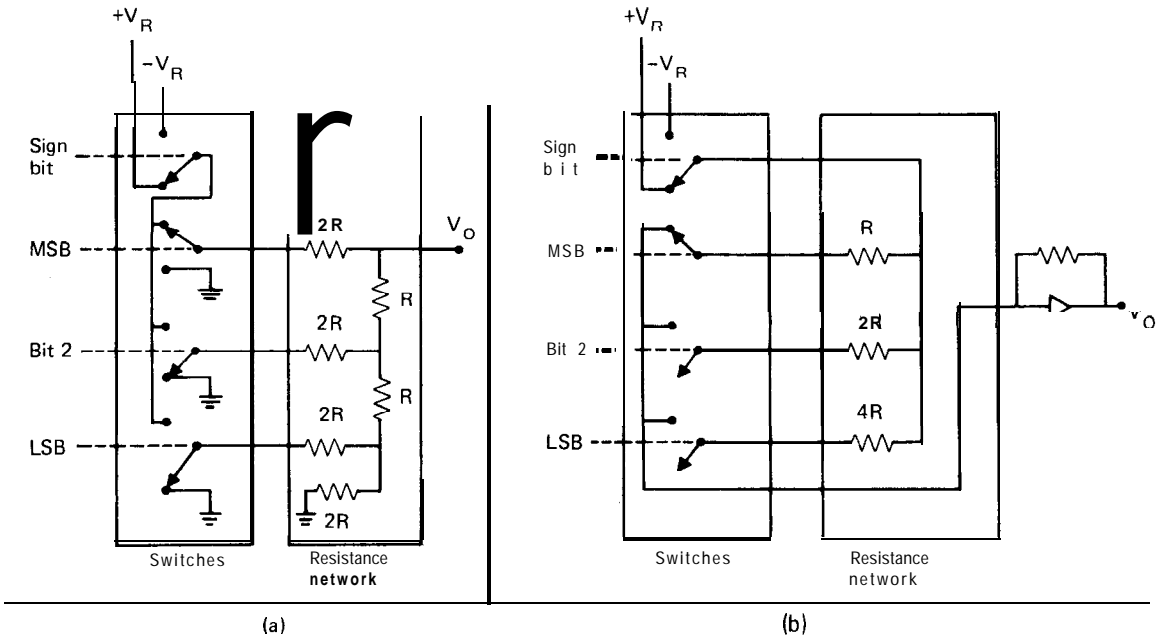


Fig. 5. Three-bit plus sign converter. (a) $R/2R$ D-A converter. (b) $R/2^n R$ D-A converter.

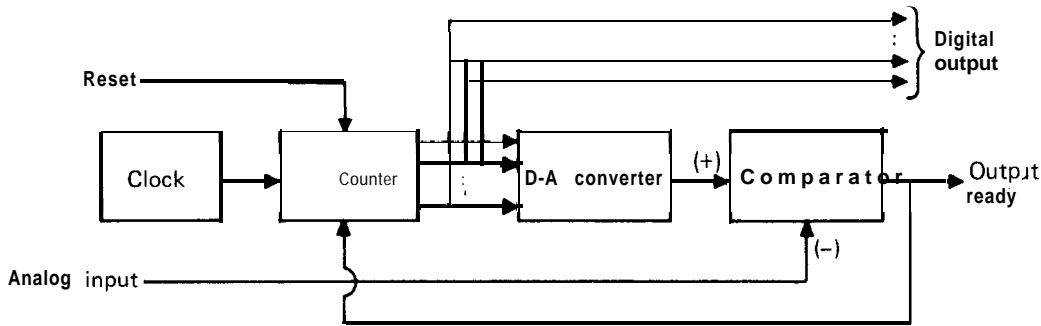


Fig. 6. Counter A-D converter.

hybrid circuit manufacturing techniques.

A-D Converters. A simple form of A-D converter is shown in Fig. 6. A conversion begins after the reset signal clears the counter. The counter now receives clock pulses and is incremented with each pulse. The counter output is a digital word representing a voltage level. This word, received by the D-A converter, results in an analog signal, which is compared with the incoming analog signal. When the comparator signal becomes positive, then the counter at that point holds the correct digital representation of the analog signal. An "output ready" signal indicates this has occurred.

The method, though simple, requires a relatively long time for a complete A-D conversion, owing to the counting process. This time increases by a factor of 2 for each additional bit and makes the method unsuitable for certain applications. A modification to the above technique, which speeds up the converter, calls for the incrementing counter to be replaced with an "up-down" counter. Here, once a comparison has been made, the counter is designed to increment or decrement on each clock pulse, depending on the output of the comparator. The counter thus follows the analog signal, and the full counting process is not necessary on each conversion if large changes in the analog input do not occur. To

DIGITAL-TO-ANALOG CONVERTERS

improve response to large input changes, additional comparators and logic may be included to allow the counter to increment and decrement by more than one unit.

A widely used and moderately high-speed A-D converter is the "successive-approximation" converter. Fig. 6 depicts such a converter if the counter box is assumed to contain a register and control logic. The converter operates by successively considering each bit position in the register and setting that bit to a "one" or a "zero" on the basis of the comparator output. The MSB is first set to a "one" with all other bit positions set to "zero." This word then enters the D-A converter and the D-A output is compared with the analog input. If the result indicates the analog input is larger, then the "one" in the MSB is kept; otherwise it is set to zero. The remaining bit positions are considered successively in the same manner and a decision is made on each bit position. After the LSB is considered, the results of conversion are found in the register. Unlike the counting method the conversion time with this method is constant for every analog input for a given converter.

While the previous two A-D conversion methods considered are both sequential in nature, all or nearly all parallel methods are available for higher speeds. The simplest method uses an analog comparator for each quantization level. One version of such a 3-bit A-D converter is shown in Fig. 7. Each comparator (C) represents a voltage level, and these levels are coded into the appropriate 3-bit code with an encoding network. Though conversion effectively requires only a single step, the cost increases rapidly with the number of bits n , since the number of comparators needed is $2^n - 1$. Other factors, such as current drawn and input capacitance, also limit the number of comparators that can be connected in parallel, and such converters are usually no more than 4 to 6 bits in length.

Numerous other converter techniques are currently in use. These are discussed in the references and in the manufacturing literature. The following section considers some basic parameters used in specifying converters.

Specification of Converters. Unfortunately, at this time only limited standardization of

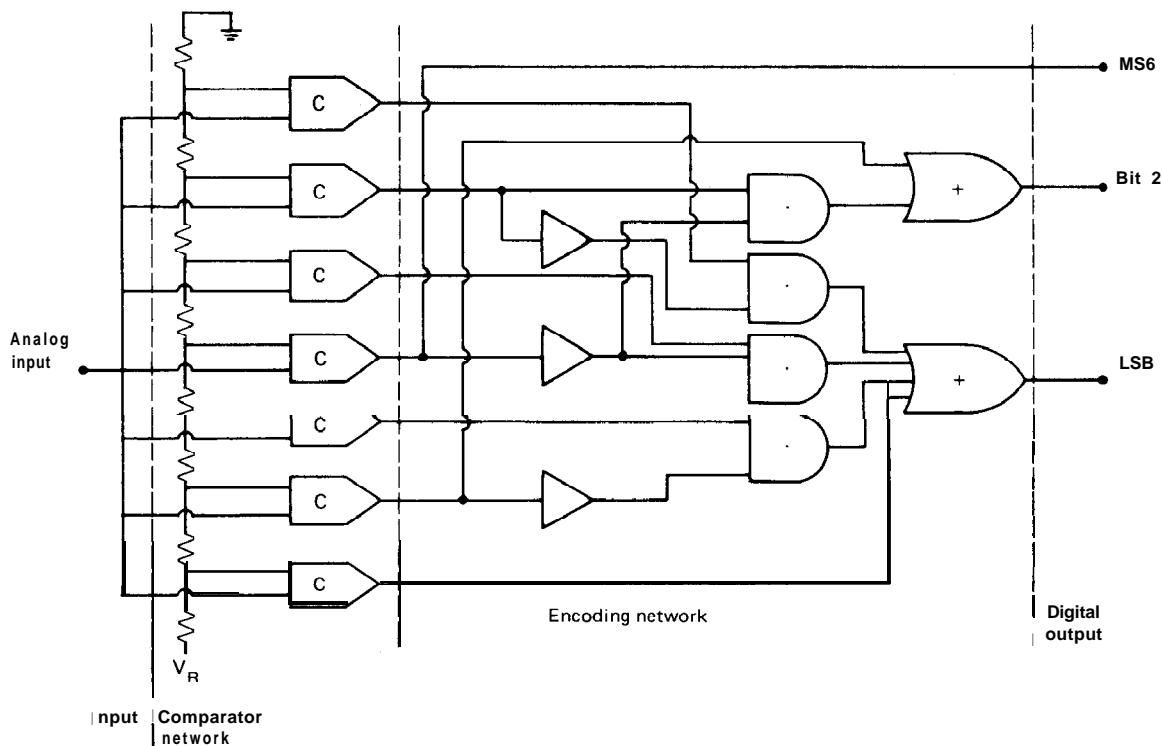


Fig. 7 Three bit parallel A-D converter.

DIGITAL-TO-ANALOG CONVERTERS

converter performance specifications has been achieved. The user should therefore be cautious in evaluating manufacturer specifications and should clearly understand the meaning of the various terms used. Many manufacturers provide literature defining their specification nomenclature.

The application for which the converter is intended should be well understood, since this will determine which of the multitude of converters available offer the best price-performance trade-offs. In addition to accuracy and speed requirements, questions regarding logic levels and codes, scale factors, reference voltages, impedance levels, power levels, temperature stability, and noise levels must be considered. These latter questions are not considered here, and the reader should consult the references for detailed information.

A number of measures are normally used in specifying converter accuracy and speed. These measures in part isolate and indicate the various

sources of error. With D-A converters, accuracy refers to the deviation of actual analog output from the theoretical output for a given digital input. Though this may vary over the range of the unit, specifications are normally given in terms of a single number representing the maximum error over the range. This may be stated as plus or minus a percentage of full scale or plus or minus a fraction of LSB.

Several common error types which contribute to a loss of accuracy are illustrated in Fig. 8. Fig. 8(a) shows *nonlinearity* in the conversion transfer function. The nonlinearity is, however, *monotonic*, since increasing digital values produce increasing analog values. Fig. 8(b) shows a *nonmonotonic* nonlinearity. Here, two different digital input codes yield the same analog value, a result that could cause oscillations to occur in certain control applications. Figs. 8(c) and 8(d) illustrate *gain* and *offset* errors, which respectively change the slope and zero crossing of the

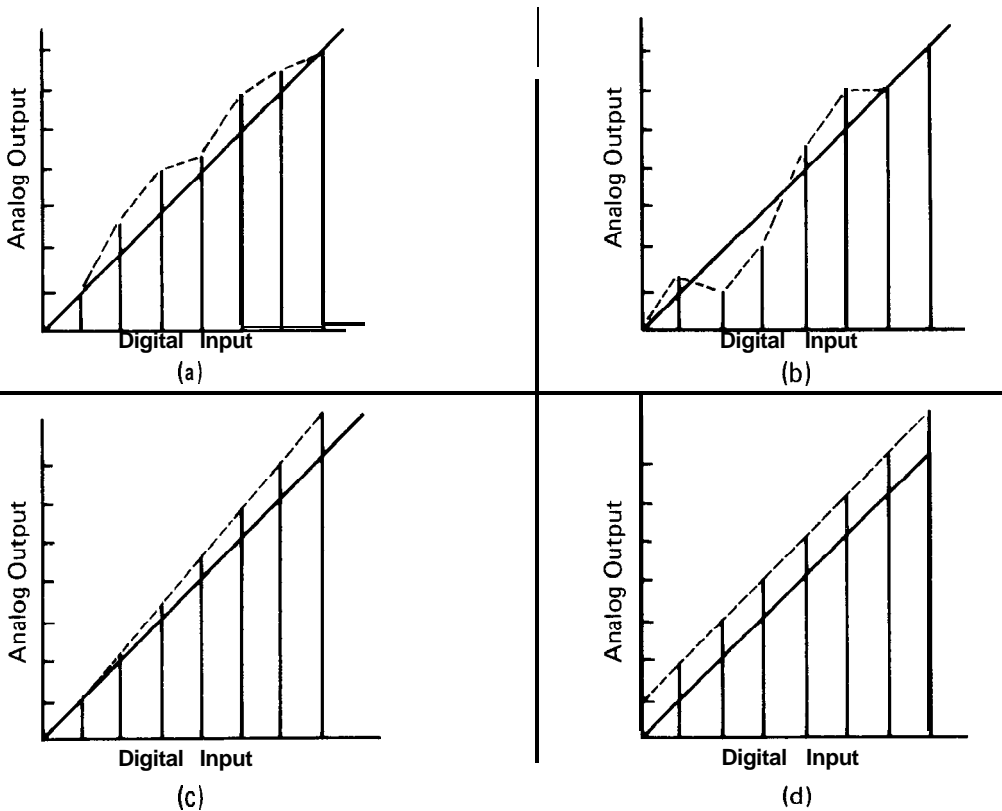


Fig. 8. D/A errors. (a) Nonlinearity. (b) Nonmonotonicity. (c) Gain error. (d) Offset error. (From D. H. Sheingold (Ed.), *Analog-Digital Conversion Handbook*, Analog Devices, Inc., with permission.)

DIRECT ACCESS

transfer function. The difference between the dotted line in the figures and the solid 45-degree angle line is the error associated with each digital input code.

Dynamic characteristics of D-A converters are normally specified in terms of a *settling* time. This is the time between arrival of the digital code and settling of the analog output to within certain specified limits of accuracy. The shorter the settling time, the higher the conversion rate. A typical specification of a moderate speed converter might read 2 μ s to settle within $\pm 1/2$ LSB.

For A-D converters, "accuracy" refers to the deviation of the analog level, represented by the digital output, from the actual analog input. As with D-A converters, this is normally stated as either a percentage of full scale or a fraction of LSB. Errors here may be divided into two parts. The first, *quantization error*, was discussed earlier in this article. This results in an inherent error of $\pm 1/2$ LSB, which can be reduced only by increasing the number of bits. All other errors are equipment errors, and error types directly corresponding to those found in D-A converters may be present. Offset, gain (scale factor), and nonlinearity errors have analogous definitions. The error corresponding to nonmonotonic nonlinearity is termed the "differential linearity error" and may result in entire digital outputs being missed.

The dynamic characteristics of A-D converters are normally specified in terms of the total conversion time. This is the time necessary for a complete measurement, its inverse being the maximum *sampling rate* of the converter. A moderate speed 8-bit converter, for example, might have a conversion time of 40 μ s and an accuracy excluding quantizing error of about 0.2% full scale.

Conclusion. A-D and D-A converters are finding increasing use as the scope of digital processing and communications widens. There is every indication that this trend will continue and will be augmented by further gains in performance and decreases in converter cost. This will result in large part from the growing use of monolithic circuit technology.

REFERENCES

1964. Stephenson, B. W. *Analog-Digital Conversion Handbook*. Maynard, Mass.: Digital Equipment Corporation.
1968. Hoeschele, D. F., Jr. *Analog-to-Digital/Digital-to-Analog Conversion Techniques*. New York: John Wiley.

1970. Schmid, H. *Electronic Analog/Digital Conversions*. New York: Van Nostrand-Reinhold.
1972. Davis, S. "Selection Criteria for A-D and D-A Converters," *Computer Design* (September).
1972. Sheingold, D. H. (Ed.). *Analog-Digital Conversion Handbook*. Norwood, Mass.: Analog Devices, Inc.

M. A. FRANKLIN

DIRECT ACCESS

For article on related subject see Memory: Auxiliary.

For articles on related terms see **FILES**; and **RAMAC**.

Early hardware, developed for the storage of large files in a data processing system, depended on two media-punched cards (in the very early days) and magnetic tapes on the early computers. Although widely different in physical characteristics, they had something in common—they forced the user to store his file records in some predetermined sequence and to process them in that same order.

Punched-card users had had no option but to accept the limitations of their equipment, but there seemed to be something quite out of balance between the short time a computer took to update a file record and all the sorting, collating, etc., necessary to find the record in the first place. Worse, although users often could learn to live with these limitations, there were certain types of commercial and industrial operations where they were quite unacceptable. In processing banking transactions, for example, it is often very desirable to be able to update each record as and when each transaction is made. In any system that calls for the interrogation of a file, such as airline seat reservations, it is obviously imperative to be able to handle each separate transaction as and when it occurs in a random sequence.

Records in magnetic tape files are stored in the sequence of some key identifier, and access to them is therefore "serial", i.e., item by item in that sequence. The terms "sequential access" and "serial access" are therefore used. What is required is a system for access directly to any desired record; the term usually given to this is "direct access."

Not until the late 1950s was suitable hardware developed to permit files to be stored in such a way that access to any desired record could be obtained

in the same time as to any other, and in an acceptably short time. The machine that first accomplished this was the IBM 305 RAMAC (**Ran-dom Access Method for Accounting and Control**), and the storage device was the magnetic disk file.

The magnetic tape unit is in many ways like the domestic tape recorder, where we often have to run through many feet of tape to find the recording we want. The magnetic disk file is in many ways like the phonograph, where the recording head can be moved very quickly (given a steady hand) to any desired position on the surface of the disk to select the desired piece of recording. In practice, the computer disk file is usually equipped with a number of disks, with a separate recording head for every disk surface. In this way, the selection of a desired record at random can usually be made in a fraction of a second, and the computer system can therefore respond in an acceptable time scale to the input item, usually in about one-third of a second.

Where higher speeds are required (measured, say, in hundredths of a second), magnetic drums (Fig. 1) are used. These are fast, rotating cylinders with file information stored on tracks along the surface and with a recording head for every track. There is no need, therefore, to move the heads (as with disk files), and the time for access to required information depends only on the speed of rotation of the drum.

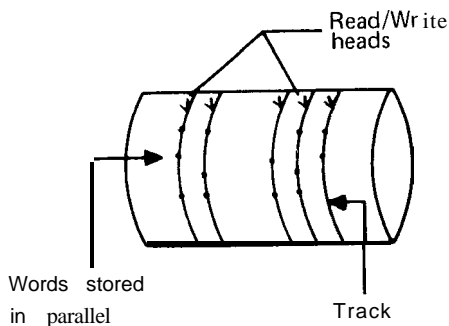


Fig. 1. Magnetic drum.

The need for speed in accessing file records is clear. What is even more obvious is the requirement for ultrarapid access to program instructions and data held in the computer's main memory. These are required with an access time measured in millionths of a second or less so that the speed of the arithmetic unit is not wasted.

It is an interesting paradox that one of the fastest devices currently used for random access to

files, the magnetic drum, was used in some of the earliest computers for the main memory and was rejected because it was too slow for random access to instructions and data! It was replaced by magnetic core memory units, which still form the majority of main memories in today's computers. These core units are capable of providing required data or instructions in times ranging from a few millionths of a second down to one-fifth of a millionth of a second, speeds compatible with those of arithmetic units. Main memories are therefore also classifiable as "random" or "direct" access.

G. J. MORRIS

DOCUMENTATION

For articles on related subjects see **ADMINISTRATIVE-BUSINESS APPLICATIONS; BLOCK DIAGRAM; FLOWCHART; FLOW DIAGRAM; STANDARDS; and SYSTEM CHART.**

For article on related term see **FEASIBILITY STUDY.**

Documentation is a vital part of developing and using a computer-based system. In some commercial organizations, 20 to 40% of the total development effort goes into the documentation of the new system, recording how the new system is to work and how it was developed. Documentation of a computer project falls into two broad categories: development documentation and control documentation. Development documentation records how a computer-based system is to operate and gives the background information upon which the design is founded. Control documentation, on the other hand, serves an administrative function: It records the resources used in developing and implementing the system, and includes such documents as project plans, schedules, resource allocation details, and progress reports.

Functions of Documentation. Documentation serves four main functions:

1. Intertask/interphase communication.
2. Historical reference.
3. Quality and quantity control.
4. Instructional reference.

DOCUMENTATION

The relative importance of each of these depends on many factors. For example, one of the most important is the scope and type of the project; it may be a large-scale commercial system, or a scientific problem-solving program used by one or two technicians on a limited amount of data. Within each category, there are the variations in project size, problem complexity, organization of staff, and the time scale for development and use. Each function of documentation is described below.

INTERTASK/INTERPHASE COMMUNICATION. This operation records what has been done at each stage of the project so that instructions can be issued for the next phase of work, or so that all people involved in the project can agree what has been done before work proceeds to the next step. The amount of time and effort that must be devoted to documentation for this reason is a function of the scope of the system and the number of people involved.

In the development of a major commercial system, which requires procedures such as invoicing, inventory control, payroll, or production control, many people will be involved. In a production control system, for example, the business functions involved could include, among others:

1. Sales forecasting (linking with sales accounts).
2. Parts explosion and production batching/netting (linked with engineering design).
3. Plant resource allocation and scheduling.
4. Materials ordering/tooling and allocation.
5. Monitoring job progress.
6. Scrap and bonus reporting (linking with payroll).
7. Job costing (linking all systems).

Most of these functions are closely related to one another and with other systems in the company. Some 20 or 30 separate job functions or organizational units may be involved with the development, implementation, and running of the computer system. In addition to job functions such as those described above, different levels of user staff will involve senior or executive management, line management, and supervisors and operators. Similarly, a number of job functions will be performed by personnel in the data processing or management services department; for example:

1. Business analysts, internal business consultants who advise management on business methods and who identify areas for improvement.
2. Systems analysts, who investigate, analyze,

and specify a new system.

3. Systems designers, who design the new system (computer and manual procedures) in detail.

4. Programmers, who design, code, and test the computer programs for the system.

5. Operators, who are responsible for the day-to-day running of the system.

There may also be general support or service staff within data processing, such as maintenance programmers, software support people, forward planning, and standards analysts. In a small installation, many of the job functions listed above may be performed by one man or a small group of men; in a large installation, each job function may be performed by a specialist group. It can be seen that keeping people informed, passing on information and ideas for approval, and giving instructions involves a complex communications network in which formal documentation plays a vital role.

Fig. 1 is a general schematic of the main lines of communication in developing a commercial computer system. Much information can pass between the various parties to keep people informed of progress, to provide sufficient information for the system design ideas to be approved, and to pass on instructions and specifications for further action.

A failure of communication through poor documentation (or a lack of it) can prove very expensive indeed. The documentation will also help to insure project continuity should staff changes occur. Experience indicates that the time span over which a system is developed and implemented (from first thoughts to live running) is increasing in many companies, and the size and complexity of commercial systems is growing. The objective of insuring project continuity despite staff changes thus becomes increasingly important.

The use of documentation for **intertask/interphase** communication is equally important in large technical or scientific projects. Where the development of a program or group of programs can be done by only a limited number of technicians, who are quite often both problem proponents and solution programmers, the importance of documentation during the project diminishes. However, the documentation of what has been done and how the programs work will be important for historical or instructional reference, as described below.

Historical Reference. The reference function is relevant to both commercial and scientific work. It is the documentation of how the system works that makes it easily changed after it is implemented. All systems are subject to change, with the sole excep-

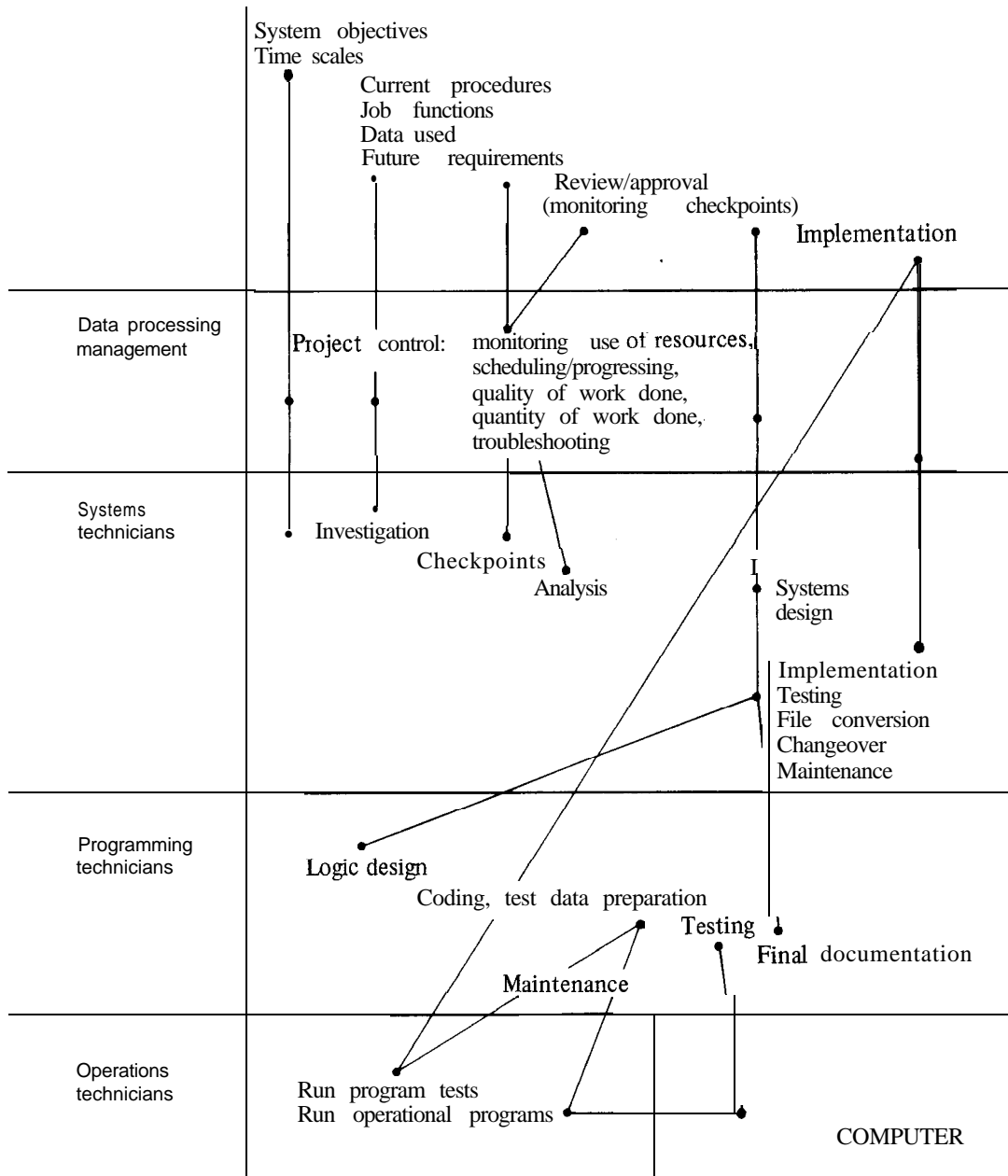


Fig. 1. Example of information flow in system development.

DOCUMENTATION

tion of one-time problem-solving applications with limited amounts of data; these are usually scientific. (One-time applications should be treated with care. Many so-called one-time applications can become repetitive, routine jobs.) Maintenance of business systems and programs will be required because the nature of a business and its methods change, or because the organization is restructured, new types of products are developed, management reporting requirements change, and so on.

In scientific work, programs may have to be altered because the nature of the problem to be solved changes, possibly as a result of further research. A system may have to be changed because of new software or hardware. It may be desirable to change the processing methods because new techniques become available. The reason for the change may lie outside the organization altogether, as is the case with legal requirements and statutory changes. The changes may be made to the system as long as five or six years after implementation.

A system can be maintained efficiently only if the existing operation of all procedures and programs is clearly known and understood. The documentation of the system provides this knowledge. For example, a program written a year ago is to be changed today; the program consists of 2,000 instructions with many branches and nested loops. The programmer who originally wrote it is no longer available. The modifications require that logic of the program be understood; the new programmer must insure that he does not introduce errors by overlooking the impact of some of his changes.

The documentation of a system may also be reviewed for performance purposes. Many installations develop performance standards based on records of time and resources budgeted and used in developing a system, as compared with system type, scope, and complexity. The control documentation is used for details of resources, and the development documentation for a description of the system. By formally capturing details of all projects, estimates of resources for future projects can be improved.

Quality/Quantity Control. As a system develops, various elements of documentation are completed as each step is finished. Management can use this documentation to evaluate project progress and individual performance.

Instructional Reference. The development documentation can be reviewed during and after development for many general purposes. For example, documentation will enable trainees to study a system developed by experienced technicians. This is particularly important for instructional reference to

generalized systems or general-purpose software. Another benefit of documentation is that an outside party can evaluate the system and its method of operation to determine if the package is suitable for use in his environment. In this case, sufficient information must be given to enable the user to apply the software to his own problems and requirements.

Instructional reference thus includes all literature provided by a software supplier, such as the reference manuals for all languages, utilities, operating systems, subroutines, and application packages. It also includes the documentation and library facilities in a large organization (such as a large decentralized company or a university) that produces its own generalized software or participates in a general interchange or pooling of programs.

Types of Documentation. Thus far, the functions of documentation have been discussed and the importance of providing it has been emphasized. In the development of a system, whether it is a large-scale commercial system or a group of scientific programs for analyzing data, certain categories of documentation must be considered. These are:

1. Analytical.
2. Systems.
3. Program.
4. Operations.
5. User/management aids.

Each of these categories is described below, and the major factors that influence the form of the documentation in any particular organization are then discussed.

1. *Analytical documentation* consists of all the records and reports produced when a project is initiated. For all projects except those that require a single, one-time, problem-solving program, some form of initial briefing is required. In most organizations, the technicians who design, program, and test a system are grouped into a computing or data processing department, and the users who commission work from the data processing department must define the nature and objectives of the project. In some technical or scientific environments, the user is capable of specifying in very exact terms what he requires in the way of processing and outputs. Generally, for any type of project, however (including many business applications), the initial briefing should consist of a user *request*, stating the problem (i.e., what the user wants to achieve); a **feasibility study** that evaluates possible solutions (in

SYSTEMS SPECIFICATION	
Title and Administrative Material	
1.0	Systems Summary
	1.1 User Summary
	1. Purpose and Function
	2. Files Maintained and Affected
	3. Input and Input Sources
	4. Output and Output Uses
	1.2 System Flowchart
	1. Flowchart
	2. Reference Lists
	1.3 Narrative Description
	1. Definitions
	2. System Flow
	3. General Timing and Size Estimates
2.0	File Specifications
	2.1 File Identification and Characteristics
	1. General Description
	2. File Abstract
	2.2 Record Format
	2.3 Data Element Descriptions
	2.4 Appendices
	1. Layouts
	2. Edit Lists
	3. Cross-Reference Lists
	cont./
3.0	Input Specifications
	3.1 Identification and Purpose
	3.2 Transaction Listing (media purpose, programs affected, frequency, volume and source)
	3.3 Input Layouts and Samples
4.0	Output Specifications
	4.1 Identification and Purpose
	4.2 Output Listing (program no., media, frequency, volume, no. of copies, and destination)
	4.3 Output Description
	4.4 Output Formats
5.0	Program (Processing) Specifications
	5.1 Program Specification 1
	5.2 Program Specification 2
	.
	.
	.
6.0	5.n Program Specification n
	Systems Test Plan
	6.1 Identification
	6.2 Test Organization
	6.3 Validity Criteria (control, processing, and output)
	6.4 Test Schedule
	6.5 Test Cases
7.0	Implementation Plan (timing, resources, responsibilities, and method)

Fig. 2. Sample outline of specification documentation. Note that items 2.0, 3.0, and 4.0 are repeated for each file. Data common to a number of programs may be defined in a Data Specification section (not shown). An added section might include a final cost-benefit analysis.

DOCUMENTATION

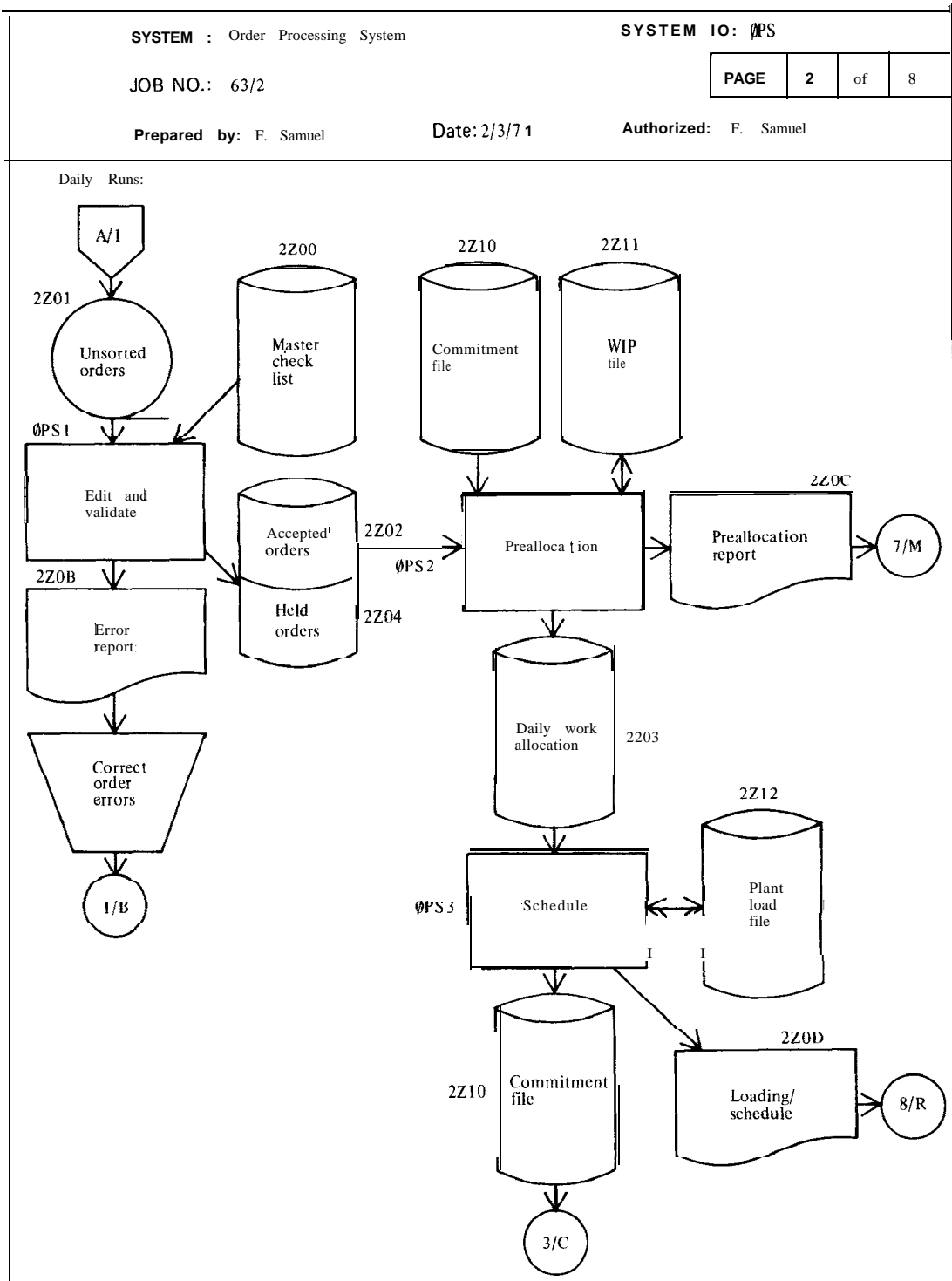


Fig. 3. Part of a system flowchart, showing the sequence of manual procedures and computer programs with their inputs and outputs. For each program box a Program Specification will be prepared. Data specifications will be prepared for all inputs, outputs, and files. See also Fig. 6.

outline); and a project *plan* that estimates the time and resources required to develop and implement the system. Failure to produce and agree upon these three statements in the briefing will result in much wasted effort later in the project. They are vital whenever a user commissions work from computer technicians, and must be provided before money is actually committed to the more time-consuming tasks of system design and programming.

2. *Systems documentation* encompasses all information needed to define the proposed computer-based system to a level where it can be programmed, tested, and implemented. The major document is some form of *system specification*, which acts as a permanent record of the structure, its functions and work flow, and the controls on the system. It is the basic means of communication between the systems design, programming, and user functions. In a major project, the system specification comprises a number of documents. A sample outline of specification documentation for a major project is shown in Fig. 2. If the project will result in the development of only one or two programs for restricted use, then only the *program (processing) specification* would be produced. Fig. 3 is an example of part of a system flowchart taken from a system specification (Section 1.2).

3. *Program documentation* comprises the records of the detailed logic and coding of the constituent programs of a system. These records, prepared by the programmer, aid program development and acceptance, troubleshooting, general maintenance, machine/software conversion at a later date, and programmer changeover.

Program documentation covers both specific applications programs and general-purpose or in-house developed software. In addition to documenting how a program works (information not always released in the case of general software), instructions for using the program must be written for packaged software (this is described in "User/management aids" below).

4. *Operations documentation* specifies those procedures required for running the system by operations personnel. It gives the general sequence of events for performing the job and defines precise procedures for data control and security, data preparation, program running, output dispersal, and ancillary operations.

5. *User/management aids* consist of all the descriptive and instructive material necessary for the user to participate in the running of the operational system, including notes on the interpretation of the output results. Where a software package is produced, this category includes all the material nec-

essary to evaluate the programs and all the instructions for use.

Every installation should establish documentation standards (i.e., rules for the completion of certain documents at certain times) that define the content, format, and distribution of the documents. Many factors influence what documents are to be produced, how, when, and by whom. For example, the extent of *management commitment* is indicated by how much the management of the installation is prepared to allocate time and resources, not only for developing a system, but also for its documentation. Another controlling factor may be *project characteristics*, which consist of the number of projects and their scope, complexity, and duration whether there are to be one or two programs operating on data from limited sources for a limited period of time, or a routine system comprising many programs operating on data from a large number of sources. Crucial to any set of standards is *the organization structure* of both the institution as a whole, and the development and operations departments in particular. This, in turn, is affected by *the technical environment*: the hardware/software techniques used, such as the level of programming language, the quality of documentation produced by the software, and the use of special-purpose documentation programs (flowcharters, etc.).

From this broad picture of the total documentation of a project, we select one type to review in detail: program documentation. We focus on this because the limits of the tasks of programming can be clearly defined, and because this function in programming is similar in many organizations.

Program Documentation. Fig. 4 shows the flow of documentation in designing, coding, and testing a program, respectively. The starting point is a program specification. Typically, this is a statement of *what* the program must do; the programmer's task is to determine how the program will do it. How much the data formats are predefined and how much is left to the discretion of the programmer depends on installation policy and the project. Other inputs to the programming phase include literature—which describes the software available for the project (either from outside suppliers or from an internal library)—and the programming standards, which give the rules and techniques for programming in that installation.

The outputs include a program manual, which describes the programs in detail (construction, coding, and testing), instructions for use (for a gener-

DOCUMENTATION

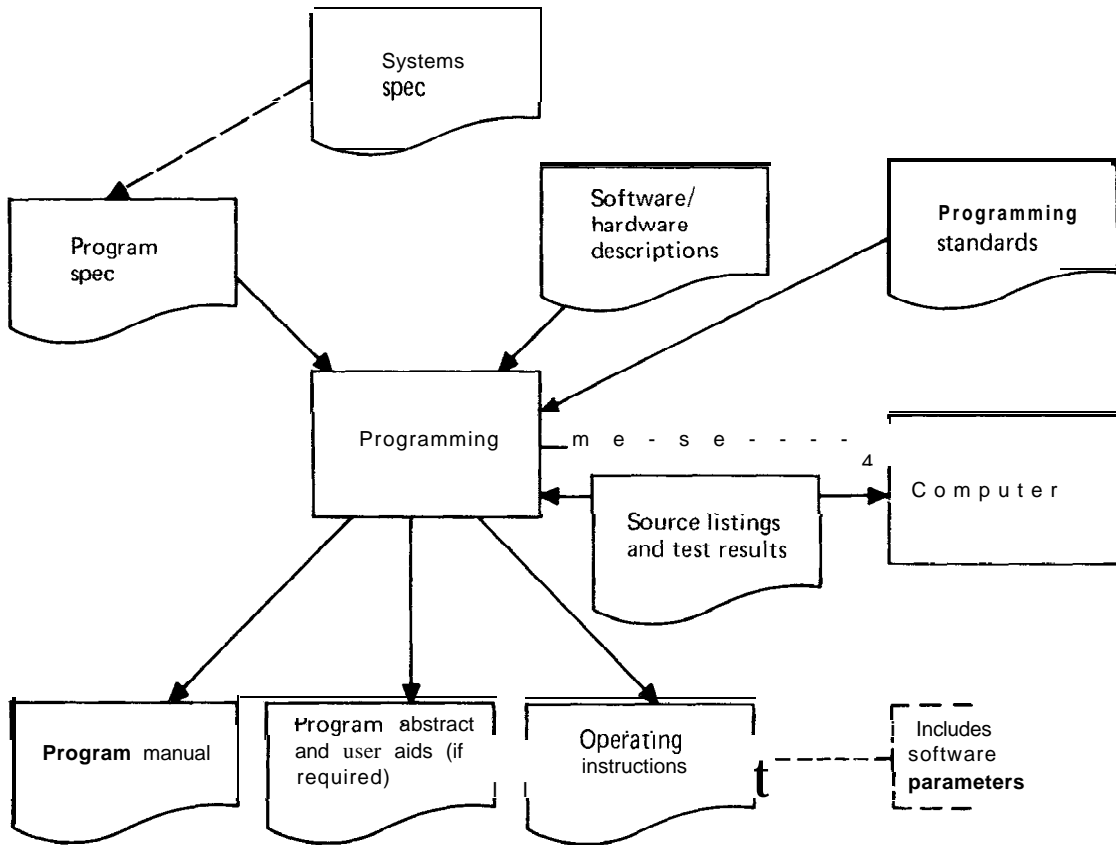


Fig. 4. Documentation flow.

alized program), and computer operating instructions for day-to-day running. In many cases the task of documenting a program is one of adding to the initial program specification in order to build up the program manual (see Fig. 5). The various elements of program documentation are discussed below.

PROGRAM SPECIFICATION. This is a statement of the data available for processing, the required outputs, and the details of the necessary processing. The specification can be prepared by the problem proponent, a specialist systems analyst/designer, or the programmer himself. It must be complete, accurate, and unambiguous; changes to the specification after programming begins can be very expensive. The specification usually contains the following information:

1. Input.
2. Output.
3. Major functions performed.
4. The means of communication between this

program and previous and following programs.

5. Logical rules and decisions to be followed, including statements of how the input is to be examined, altered, and utilized.

6. Validation and edit criteria.

7. Actions to be taken on error or exception conditions.

8. Special tables, formulas, and algorithms.

(Where a utility program or application package is being used, then some of the data listed will be omitted, and parameters specifically related to the program will be listed instead.) The description of the processing rules (item 5 in the list), can be given in narrative, flowchart, or decision-table form. Figs. 6(a), (b), (c) show the components of a program specification.

PROGRAM MANUAL. From the program specification, the programmer designs, codes, and tests the program. The output of this exercise is the program manual (see Fig. 5). A flowchart from such

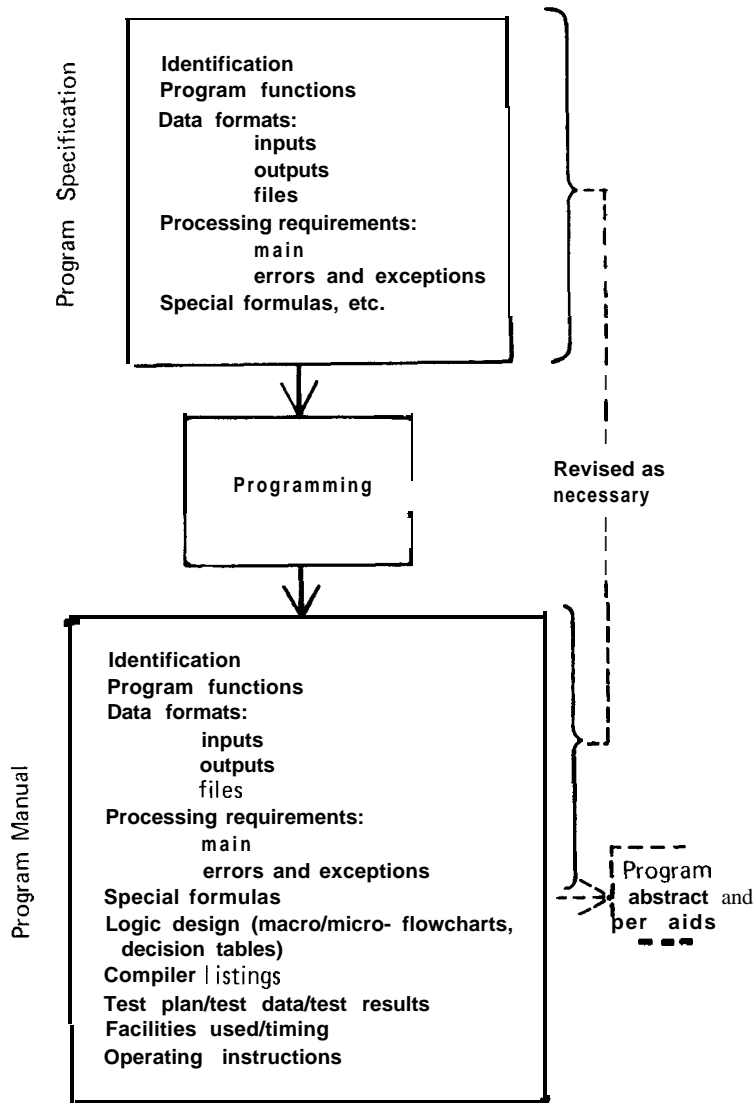


Fig. 5. Program specification and program manual.

DOCUMENTATION

SYSTEM: <i>Order Processing System</i>	SYSTEM ID: <u>o PS</u>
FROG RAM. <i>Order Edit and Validate</i>	PROGRAM ID: <u>ØPS1</u>
Prepared by: <i>J. ROBERTS</i>	Date: <i>4/5/71</i>
Authorized: <i>I. Samuel</i>	

PROCESS CHART

```

graph TD
    1((1)) --> Edit[Edit and validate]
    2[(2)] --> Edit
    3[(3)] --> Edit
    Edit --> 5[/5/]
    
```

INPUT/OUTPUTS		INPUT/OUTPUTS	
File Name	ID	File Name	ID
1. Unsorted orders	2201	4. Held orders	2204
2. Master check list	2200	5. Error report	220B
3. Accepted orders	2202		

(a)

File Specification		FILE ID: 2209	
File name: Master check list			
Prepared by: J. Roberts		Date: 4/5/71	
		Authorized: F. Samuel	

Medium:	Disk
Contents (record names):	Order Identity, Part Request, End Record
Sequence (if any):	Key field within record type
Retention and Protection:	Master
Used On (program ID's):	

File Description:

Type of record		Organization	
Blocked	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto; display: flex; align-items: center; justify-content: center;">✓</div>	Sequential	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto;"></div>
Unblocked	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto;"></div>	Direct Access	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto;"></div>
Fixed length	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto; display: flex; align-items: center; justify-content: center;">✓</div>	Index Sequential	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto; display: flex; align-items: center; justify-content: center;">✓</div>
Variable length	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto;"></div>		
Undefined length	<div style="border: 1px solid black; width: 40px; height: 40px; margin: 0 auto;"></div>		

Sizes (bytes)	Average	Maximum	Minimum
Block length:		1232+4	
Record length:		52+4	-
File length:	2,300	4,000	1,400

Remarks:

Used to validate all input order requests by order identity and part characteristics.

(b)

[illegible]

(c)

Fig. 6. Sample documents from a program specification. (a) The process chart identifies the system and the program, and shows inputs and outputs. (b) Sheet 1 of the file specification describes the **file** as a whole. (c) Sheet 2 of the file specification describes the content and format of a record. Note the use of preprinted forms and the highly stylized, rigid method of completion.

a program manual is shown in Fig. 7. The form of the logic design and the source program listing will depend on the type of application and the software used. For example, if a high-level decision-table preprocessor is used, the tabular program together with the data descriptions and final source listing will be complete enough without the preparation of a flowchart. Similarly, some installations use software for the final documentation; e.g., flowcharters that produce detailed flowcharts, statement by statement from the source program.

One of the advantages of higher-level languages (such as Cobol) is that the source listing itself forms

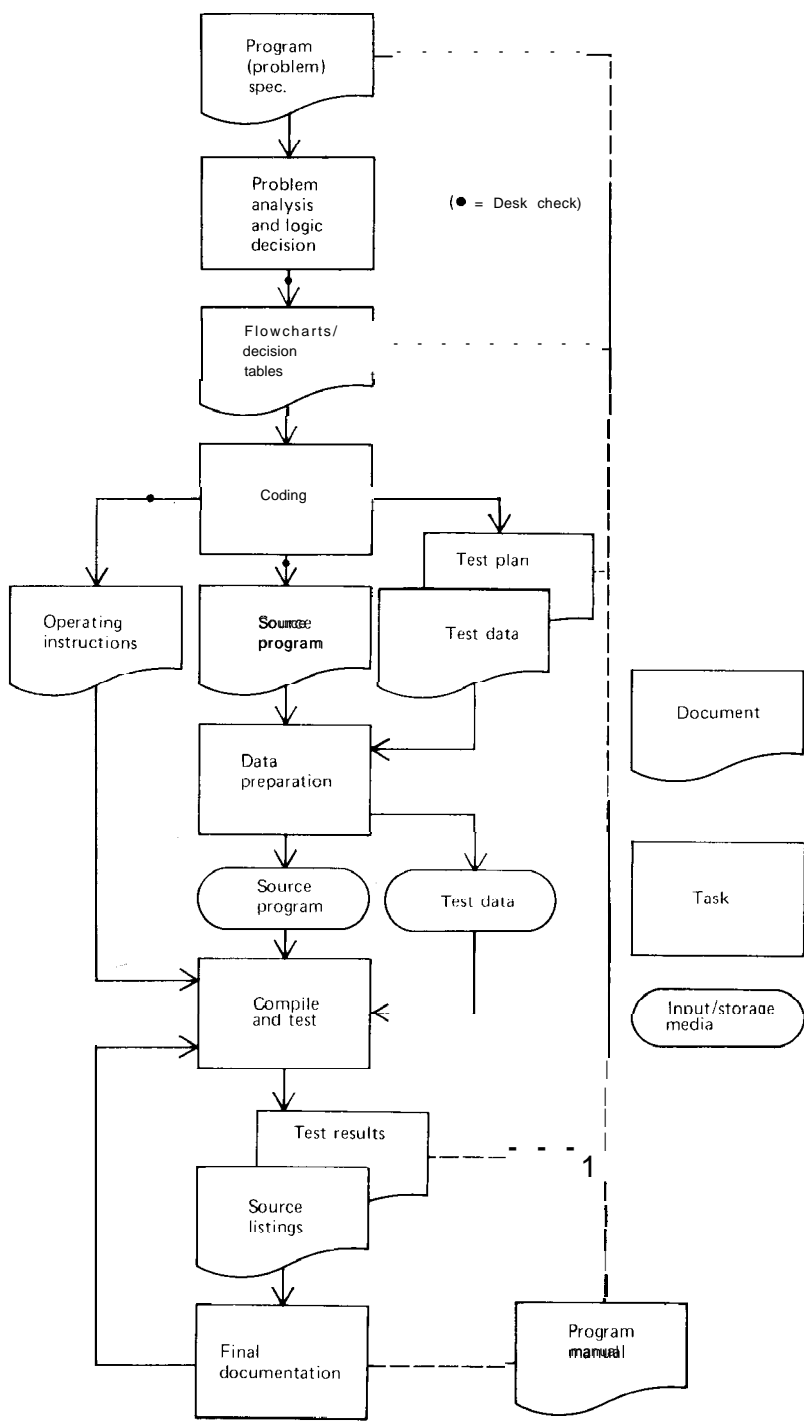


Fig. 7. A flowchart from a program manual.

DOCUMENTATION

the major part of final documentation. The programmer must insure that the source program not only is logically correct and follows the rules of the language, but also that the program coding is neat and easy to understand. By using meaningful data names and comments embedded in the program, it is possible to make a source program almost **self-explanatory**. In this case, a flowchart can be used as a general "route map" to the detailed coding in the source listing. When altering an operational program, many programmers refer directly to the source listing and then to the flowchart only if the required change is not immediately obvious from the listing.

Flowcharts and cross-reference lists (produced by the compiler or other software) can be used to check that an alteration has not erroneously disturbed other coding. The advantage of using comments in the source listing is that it minimizes references to other documents. The comments are not compiled as part of the program but merely appear on the source listing. Source program comments should be kept brief while at the same time being descriptive and meaningful.

Note that the final documentation will show not only how the program works, but also how it was tested (for quality control and later retest after changes are made), operating instructions for running it, and any special parameters that are to be given to the operating system.

Although the program manual is the major output of the program specification, the programmer will use (and can produce) other types of documentation. If the program(s) being developed are for general use, either within or outside the organization, then additional user instructions will be needed. They should enable the prospective user to answer the following questions:

1. What does it do?
2. Do we want to use it?
3. Can we use it?
4. How do we use it?
5. What do we do if it changes?
6. What are its basic limitations?

For internally produced software, one approach is to produce a program abstract which can be held in a central documentation library. If, on first inspection, the user feels that it fulfills his needs, he can then consult the detailed documentation. The form of this detailed documentation depends on the scope and complexity of the software. For example, a user's guide may be produced to give a general description of the program(s), the facilities available,

the hardware environment required, and example uses of the software. The user's guide may contain instructions for using the software, or a programmer's reference manual may be supplied giving detailed information. Most software is constructed so that the user programmer supplies parameters for a particular job. The parameters may be as simple as specifying an address of data to be processed or as comprehensive as a complete list of processing requirements. The programmer's reference manual will describe the construction of the program(s), together with all parameters required (their format, interdependence, and usage), operating instructions, and error conditions and diagnostics.

Programs ready for operational use (both applications programs and software programs) and which are to be run repeatedly are assigned to the automated program library, usually stored on magnetic drum, disk, or tape. The documentation for these programs is usually held in some form of central records library, together with the master reference copies of all software descriptions.

Documentation Maintenance and Control. Once a program has been implemented, the documentation must be retained for subsequent reference. When the system is changed, the documentation will be consulted and altered accordingly. It is vital to revise the documentation so that it completely and accurately reflects the operation of the system at all times. If the documentation is not so revised, then further maintenance will be very difficult. After any major amendment, all affected programs will have to be retested to prove that the changes have been made correctly and that they do not disrupt or invalidate other processing.

It is necessary, therefore, to insure that the appropriate control procedures are used. All changes should be properly recorded and all copies of the documentation updated. There is a strong case here for restricting the number of copies of the documentation to reduce the time spent in revising records and to minimize the risk of out-of-date copies being used by mistake. This applies not only to own-produced programs, but also to generalized software descriptions used by all the programmers. All copies should show current parameter requirements for the operating system, language rules, limitations and parameters for utility programs, and operating error messages produced by all software programs. A large installation will not only create a central records library but also appoint a full-time librarian to cope with amendment distribution control. This is sometimes handled by a "software

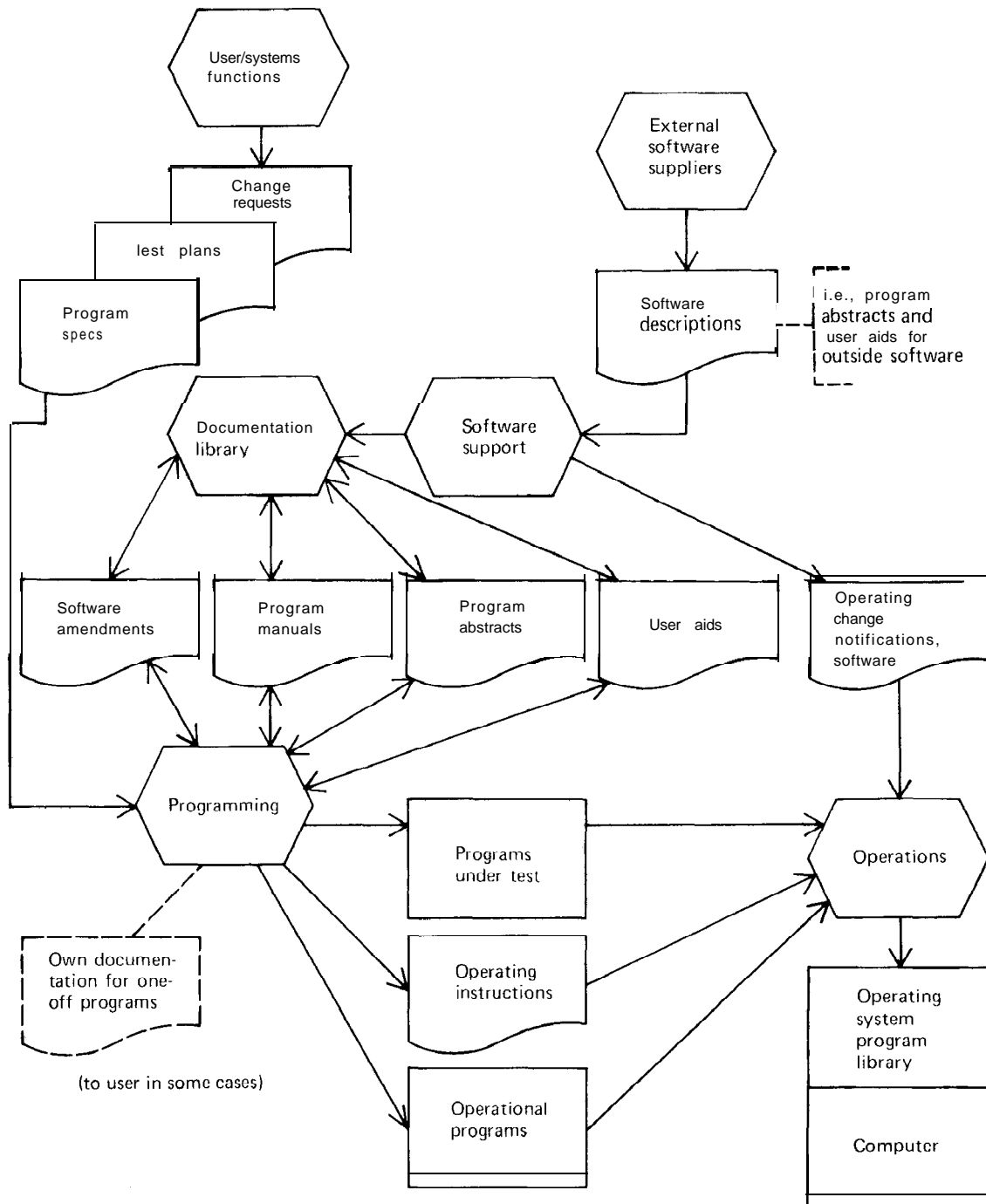


Fig. 8. Library and control program documentation. In addition, a documentation library may also accept and control systems documentation.

DUMP

support" department, which will insure that both programming and operations departments are informed of changes in software availability and operation, such as the introduction of a new release of the operating system. An example of how program documentation will be handled in a large installation is shown in Fig. 8.

Summary. Documentation is a vital element in developing and running any computer project, whether in a government, business, academic, or military installation. It must not be handled in a haphazard fashion; formal documentation standards must be laid down and enforced*. These standards must cover all areas: users, systems, and programming and operations. In a modern computer installation the flow of documentation can be complex, encompassing in-house systems and programs as well as externally produced software.

REFERENCES

1963. **Brandon, D.** *Management Standards for Data Processing*. Princeton, N.J.: Van Nostrand.
1972. **Van Duyan, J.** *Documentation Manual*. Philadelphia: Auerbach.
1973. **London, K.** *Documentation Standards* (rev. ed.). Philadelphia: Auerbach.

K. R. LONDON

DUMP

For articles on related subjects see **DEBUGGING**; and **MACHINE AND ASSEMBLY LANGUAGE PROGRAMMING**.

For article on related term see **SOURCE PROGRAM**.

A dump is a printed representation of the raw content of a computer storage device, usually main memory, at a specified instant. "Raw" means that little or no interpretation is performed on the content; it is taken simply as a number of bit strings and presented to the reader as such.

A few refinements are found in even the simplest dumps that keep them from being mere one-to-one bit maps: The representation is usually in octal or hexadecimal, reducing the dump's bulk by a

factor of 3 or 4. The segmentation of memory into words or bytes is reflected in the print format; the address of the leftmost word or byte on each printed line is given; and long stretches of identically filled memory segments (typically, zero-filled) are not printed verbatim, but replaced with a message like **LOCATIONS 4000-4 177 ALL ZERO**. Simple dumps often offer the further amenities of permitting bounds to be set on the area of storage to be printed, and of automatically including the contents of the chief registers in the CPU.

The dump is the most primitive of debugging devices. It corresponds in vintage and sophistication to machine-language programming, and its use in debugging higher-level language programs should be nil, but unfortunately is not. While it is far more commonly employed in debugging assembly-language programs than those written in higher-level languages, it is still the last resort for programs of all descriptions, including those in higher-level languages. Its total replacement by tools of greater power and convenience has long been expected, but accomplished-if at all-only where debugging is routinely done on line. Even there, it can be argued that the dump is often not so much eliminated as disguised.

On-line debugging sessions do not involve extensive dumps, nor are the minidumps that are involved so bit-oriented, but the representations of memory contents that are produced share the dump's essential characteristic of being instantaneous snapshots of a moving object, and of requiring the programmer to shift into another language, almost into another discipline, when debugging. (Another major family of debugging aids, known collectively as the "trace," escapes one of these shortcomings, that of being a static observer of a moving object, but not the other.)

The crudeness of debugging with the sole aid of dumps, program listings, and mother wit is due not merely to the dump's being a record of a single instant only, but to its being, usually, a record of the wrong instant. By the time an observer, human or programmed, has detected something wrong with a running program and **ordered** a dump taken, it is probable that some or all of the evidence that would enable the programmer to find the underlying bug has been erased or changed.

The total replacement of the dump-or, what is equivalent, the realization of "source-language debugging" - has proved to be more difficult to achieve than had initially been expected, and may still be a long time in coming. It may require the abandonment of the notion of "debugging"-i.e., curative,

after-the-fact treatment of faulty programs-in favor of preventive or prophylactic approaches like those suggested in the references given below.

REFERENCES

1965. Halpern, M. I. "Computer Programming: The

Debugging Epoch Opens," *Computers and Automation* (November).

1971. Worley, W. S. "Toward Automatic Debugging of Low Level Code," IBM Technical Report TR 00.22 1971.

M. **HALPERN**

EBCDIC

For articles on related subjects see ASCII;
CODES; and IBM CARD.

The Extended Binary Coded Decimal Interchange Code (EBCDIC) was developed by IBM for use on the IBM System/360; it is also used exclusively on the IBM System/370. In order to remain compatible with IBM equipment, many other computers also use EBCDIC. The only 8-bit code that is a competitor to EBCDIC for use on computers is ASCII-8.

Fig. 1 shows the 256 ($= 2^8$) combinations for EBCDIC, many of which are currently unassigned but may be assigned later as new developments in computer technology occur. The leftmost four bits (or first hexadecimal digit) of the 8-bit code are shown across the top of the Fig. 1 and the rightmost four bits (or second hexadecimal digit) in the second column on the side. Table 1 gives an example.

Also shown in Fig. 1 are the punches on an IBM card corresponding to each of the characters of the code. Zone punches (12, 11, 0, and occasionally 9) for characters above (below) the heavy black lines are shown at the top (bottom). Digit punches (1 to 9) for characters to the left (right) of the heavy black line are shown on the left (right). Table 2 gives an example.

Table 1

Symbol	Code	
	Binary	Hexadecimal
4	11110100	F4
Y	11101000	E8
c	10000011	83
=	01111110	7E

Table 2

Character	Card Punches
4	4
Y	0-8
=	12-0-3
IL	11-9-8-6

The meanings of the control characters and special graphics, and the card-punch patterns of characters that do not conform to the rules above are shown in Fig. 1.

I. FLORES

EBCDIC

Bit Positions 0,1		Bit Positions 2,3		Hexadecimal		Digit	
00		01		10		11	
00	01	10	11	00	01	10	11
0	1	2	3	4	5	6	7
12				12	12		12
	11			11	11	11	
		0		0	0	0	
9	9	9	9	9	9	9	9
0000	0	8-1	1	2	3	4	5
	NUL	DLE	DS		SP	&	
0001	1	1	SOH	DC1	SOS		
0010	2	2	STX	DC2	FS	SYN	
0011	3	3	ETX	TM			
0100	4	4	PF	RES	BYP	PN	
0101	5	5	HT	NC	LF	RS	
0110	6	6	LC	BS	ETB	UC	
0111	7	7	DEL	IL	ESC	EOT	
1000	8	8	CAN				
1001	9	8-1	EM				
1010	A	8-2	SMM	CC	SM		
1011	B	8-3	VT	CUI	CU2	CU3	
1100	C	8-4	FF	IFS	DC4	<	
1101	D	8-5	CR	IGS	ENQ	NAK	
1110	E	8-6	SO	IRS	ACK	+	
1111	F	8-7	SI	IUS	BEL	SUB	
12				12			
	11			11			
		0		0	0	0	0
9	9	9	9				

Card	Hole	Patterns
1		12-0-9-8-1
3		12-11-9-8-1
4		12-11-0-9-8-1 11-0-9-8-1

	No	Punches
0		
6	12	
7	11	
8	12-11-0	

0	12-0
11	11-0
12	0-0

013	O-1
014	1 1-O-9-1
015	12-11

Control Character Representations

ACK	Acknowledge	EOT	End of Transmission
BEI	Beil	ESC	Escape
BS	backspace	ETB	End of Transmission Block
BYP	bypass	ETX	End of Text
CAN	Cancel	FF	Form Feed
c c	Cursor Control	FS	Field Separator
CR	Carriage Return	HT	Horizontal Tab
CUI	Customer Use 1	IFS	Interchange File Separator
CU2	Customer Use 2	IGS	Interchange Group Spomitor
CU3	Customer Use 3	IDS	Idle
DC1	Device Control 1	IRS	Interchang Record Separator
DC2	Device Control 2	IUS	Interchange Unit Separator
DC4	Device Control 4	LC	Lower Case
DEL	Delete	LF	Line Feed
DL	Data Link Escape	NAK	Negative Acknowledge
D5	Digit Select	NL	New Line
EM	End of Medium	NUL	Null
ENQ	Enquiry		

Special Graphic Characters

¢	Cent Sign	Minus Sign, Hyphen
£	Pound Sign	Slash
¤	Dollar Sign	Comma
¥	Yen Sign	Percent
€	Euro Sign	Underscore
+	Plus Sign	>
×	Logical AND	>
÷	Division Sign	Greater-than Sign
∆	Delta Sign	?
!	Exclamation Point	Question Mark
∂	Partial Differential Sign	Colon
∞	Infinity Sign	#
°	Degree Sign	Number Sign
•	Bullet Sign	At Sign
◊	Lozenge Sign	Prim, Apostrophe
◊	Lozenge Sign	Equal Sign
◊	Lozenge Sign	Quotation Mark

Fig. 1. **EBCDIC** code combinations.

ECKERT, J. PRESER

For articles on related subjects see ENIAC;
MAUCHLY, JOHN W.; and UNIVAC I.

J. Presper Eckert, co-inventor of ENIAC, was born in 1919 in Philadelphia. He received a Bachelor of Science degree in electrical engineering from the University of Pennsylvania's Moore School of Electrical Engineering in 1941, and his Master's degree under a graduate fellowship from the Moore School in 1943.



Fig. 1. J. PRESER ECKERT.

Dr. Eckert collaborated with Dr. John W. Mauchly, of the Moore School's staff, on developing ENIAC (Electrical Numerical Integrator and Computer) for Army Ordnance between 1943 and 1946. This was the world's first all-electronic digital computer, and could perform 5,000 additions or subtractions per second. Its development launched the computer industry as we know it today.

In 1947, Dr. Eckert and Dr. Mauchly incorporated their venture as the Eckert-Mauchly Computer Corporation. They developed BINAC, the first electronic and fully self-checking computer, in 1949. Their next project, UNIVAC (Universal Automatic Computer), was well under way when Remington Rand acquired the Eckert-Mauchly firm in 1950.

Dr. Eckert became director of engineering for Remington Rand's Eckert-Mauchly Division, which completed UNIVAC I. He became vice-president and director of research in 1955, vice-president and director of commercial engineering in 1957, vice-president and executive assistant to the general manager in 1959, and vice-president and technical advisor to the president of Sperry-Rand, UNIVAC division, in 1963.

Dr. Eckert received an honorary degree of Doctor of Science in Engineering from the University of Pennsylvania in 1964.

In 1969, Dr. Eckert was awarded the National Medal of Science, the nation's highest award for distinguished achievement in science, mathematics, and engineering.

A Fellow of the Institute of Electrical and Electronics Engineers, and a member of the National Academy of Engineering, Dr. Eckert is listed as the inventor or co-inventor on 87 patents.

M. M. MAYNARD

ECKERT, WALLACE J.

For article on related subject see **DIGITAL COMPUTERS**: Early.

Wallace John Eckert, was born in Pittsburgh, Pa., June 19, 1902 (d. Englewood, N.J., Aug. 24, 1971). Much of the credit for the introduction of machine computation into astronomy belongs to him. The significance of the computer impact on astronomy is comparable to that of the introduction and use of the telescope and photography.

Eckert was raised on a farm in Albion, Pa., the second of four boys born to John and Anna (Heil) Eckert. He received his A.B. degree from Oberlin College in 1925 and his M.A. from Amherst in 1926. In 1931, he was awarded his Ph.D. in astronomy by Yale University. He joined the Columbia University Department of Astronomy as an assistant instructor in 1926.

In 1928, Professor Ben Wood formed the Columbia University Statistical Bureau using punched-card equipment donated by Thomas Watson, Sr., of IBM. It was here that Eckert was first exposed to the possibility of using machines to facilitate computation. From 1929 to 1933 he used the machines in Prof. Wood's laboratory for the interpolation of astronomical data, the reduction of

ECONOMIC APPLICATIONS

observational data, and the numerical solution of planetary equations. In 1933, with the encouragement of Ben Wood, he convinced Watson to install punched-card equipment and a control unit for astronomical calculations. This led to the formation of the T. J. Watson Astronomical Computing Bureau, jointly operated by Columbia, IBM, and the American Astronomical Society (1937-1945). During this period he published his landmark work (1940), "Punched Card Methods in Scientific Computation."

He was director of the U.S. Nautical Almanac Office in Washington, D.C., from 1940 to 1945. He introduced machine methods to data handling in the Naval Observatory as well as the Almanac Office. During the war he designed the "American Air Almanac," a great navigational influence that is still in use with only minor modifications.

In 1945 he was appointed head of IBM's Pure Science Department and became director of the Watson Scientific Computing Laboratory. The Laboratory not only performed needed computations, but also provided a training ground in machine computation for more than a thousand scientists in crystallography, geology, chemistry, statistics, optics, and solid-state physics, as well as astronomy.

Eckert was instrumental in the construction of IBM's Selective Sequence (Electronic) Calculator (SSEC, 1949) and the Naval Ordnance Research Calculator (NORC, 1954). Using the SSEC, Eckert, Dirk Brouwer of Yale, and G.M. Clemence (1951) of the U.S. Naval Observatory computed the precise positions of Jupiter, Saturn, Uranus, Neptune, and Pluto for the period 1652-2060. This work still serves as the Ephemeris predictions for these planets.

Eckert's most important purely astronomical contributions were in relation to the moon's orbital motion. This and later work in the area of lunar coordinates and orbital parameters (1966) provided the operational basis for NASA's Surveyor, Lunar Orbiter, and Apollo projects.

He retired from IBM in 1967 and as Professor of Celestial Mechanics at Columbia in 1970.

REFERENCES

- Anon., "Dr. Wallace J. Eckert" (publications by W. J. Eckert, 38 items), and "Outstanding Contribution Award Report" (n.d.), IBM Archives.
1940. Eckert W. J. *Punched-Card Methods in Scientific Computation*. New York: Columbia University Press.
1951. Eckert, W. J., D. Brouwer, and G. M. Clemence. "Coordinates of the Five Outer Plan-

ets, 1653-2060," *Astronomical Papers, NORC*, Vol. 12, U.S. Government Printing Office.

1966. Eckert, W.J. "Transformations of the Lunar Coordinates and Orbital Parameters," *Astronomical Journal* (June),

1971. J. A. "A Great American Astronomer," *Sky and Telescope* (October).

H. S. TROPP

ECONOMIC APPLICATIONS

For articles on related subjects see **MODELS; OPERATIONS RESEARCH; and SIMULATION**.

Economics is concerned with the interrelated questions of the formation of income, the distribution of income, and the spending of income. Adam Smith's *Wealth of Nations*, 1776, is the first systematic treatment. Alfred Marshall's *Principles of Economics*, 1890, contains the first truly satisfactory explanation of value and the first systematic treatment of price theory.

In 1936, spurred by the depression, J. M. Keynes published his influential *General Theory*. The **scope** and even the purpose of economics broadened as it became recognized that it belonged to the legitimate domain of governmental responsibility to guide the economy along a full-employment, full-capacity, stable-prices, adequate-growth path, with balance of payments equilibrium and equitable distribution of income as well.

A *major step forward toward this goal of controlling the business cycle was Tinbergen's League of Nations study, *Statistical Testing of Business Cycles*, 1939. This was the first systematic attempt to portray the workings of the economy in the form of numerically specified models, i.e., systems of simultaneous equations. Economic theory provides the **basis** for such relations in *qualitative* terms; **statistical time** series and econometric estimation techniques (sophisticated versions of regression analysis) provide the **quantitative** specification. The quantitative model will, in general, include variables that the government can influence (taxes, for example) to achieve desired objectives.

Both the estimation procedures and the problem of choosing the appropriate values for the decision variables require vast computations. The most complete model of the United States economy has around 200 equations. One such equation might be

that which purports to explain yearly consumption (C) as a function of wages (W) and nonwage income (P) in that year as

$$C_t = \alpha_0 + \alpha_1 P_t + \alpha_2 W_t + u_t \quad (t = 1946, \dots, 1974).$$

The α coefficients have to be estimated. Even the "best" estimates for the α components will, for most or all years, leave a discrepancy, indicated in the equation by u_t . A frequent definition of "best" is that the sum of the squared u values ($\sum u_t^2$) is minimized. The parameters of an equation that forms part of a system of equations can be properly estimated only by simultaneous estimation techniques, and require full specification of the rest of the model and extensive computations.

A more or less parallel development began with the publication, in 1944, of Vassily Leontief's *Structure of the American Economy*, introducing the concept of input/output models. Input/output models focus on interindustry deliveries and are designed to determine how a decision in some industry reverberates through the industrial complex: how much steel does a new aircraft carrier require, and how much coal is needed to produce that steel; how many miners to produce that coal, and how many eggs to keep those miners healthy, . . . , and so on. The United States input/output model distinguishes about 80 different industries. As with econometric models, the gathering of data, the statistical estimation and verification, and the manipulation of the model to answer specific questions are voracious consumers of computer time. Without computers, the useful applications of input/output modeling would be impossible.

A more recent development is the widespread use of simulation studies to predict future scenarios for economic systems. The Club of Rome's *Limits to Growth* (Meadows, 1972) is the prototype of this new area. A special Dynamo language has been developed for his purpose. Similar simulation techniques are also widely used in operations research and management science.

REFERENCES

1965. Theil, H., et al. *Operations Research and Quantitative Economics*. New York: McGraw-Hill.
 1972. Maisel, H., and A. Gnugnoli. *Simulation of Discrete Stochastic Systems*, S.R.A., 1972.
 1972. Meadows, D. L., et al. *The Limits to Growth*. New York: Universe Books.

J. C. G. BOOT

EDITOR. See LINKAGE EDITOR.

EDSAC

For articles on related subjects see **DIGITAL COMPUTERS**: Early, and Origins; **EDVAC**; **ENIAC**; **ULTRASONIC MEMORY**; and **WILKES**, M. V.

For articles on related terms see **ECKERT**, J. **PRESPER**; **MAUCHLY**, JOHN W.; and **MEMORY**: Main.

The **EDSAC** (Electronic Belay Storage Automatic Calculator) was designed according to the principles expounded by J. Presper Eckert, John W. Mauchly, and others at the summer school held in 1946 at the Moore School of Electrical Engineering in Philadelphia, and which the author of this article was privileged to attend. The objectives in mind from the beginning were (1) to show that a binary stored-program computer could be constructed and operated; (2) to make a start with the development of programming techniques, even then seen to be a subject of more than trivial content; and (3) to apply the techniques developed in a variety of application fields.

In order to accelerate the attainment of the first objective, it was decided to ease the circuit design problems by choosing a conservative pulse repetition frequency (500 kc/s, compared with 1 Mc/s used in most contemporaneous projects) and to bias the logical design in the direction of simplicity rather than speed. This policy was successful, and by May 1949 the project had reached the stage at which the development of programming techniques and the running of practical programs could begin.

The **EDSAC** (Fig. 1) was a serial binary computer with an ultrasonic memory. The mercury tanks used for the main memory were about 1 1/2 meters long and were built in batteries of 16 tanks. Two batteries were provided. A battery, with the associated circuits, could store 256 numbers of 35 binary digits each, one being a sign digit. An instruction occupied a half-word and it was also possible to use half-words for short numbers. Numbering of the storage locations was in terms of half-words, not full words. The instruction set was of the single-address variety, and there were 17 instructions. Multiplication was included, but not division. Input and output were by means of five-channel punched-paper tape. The input and output orders provided

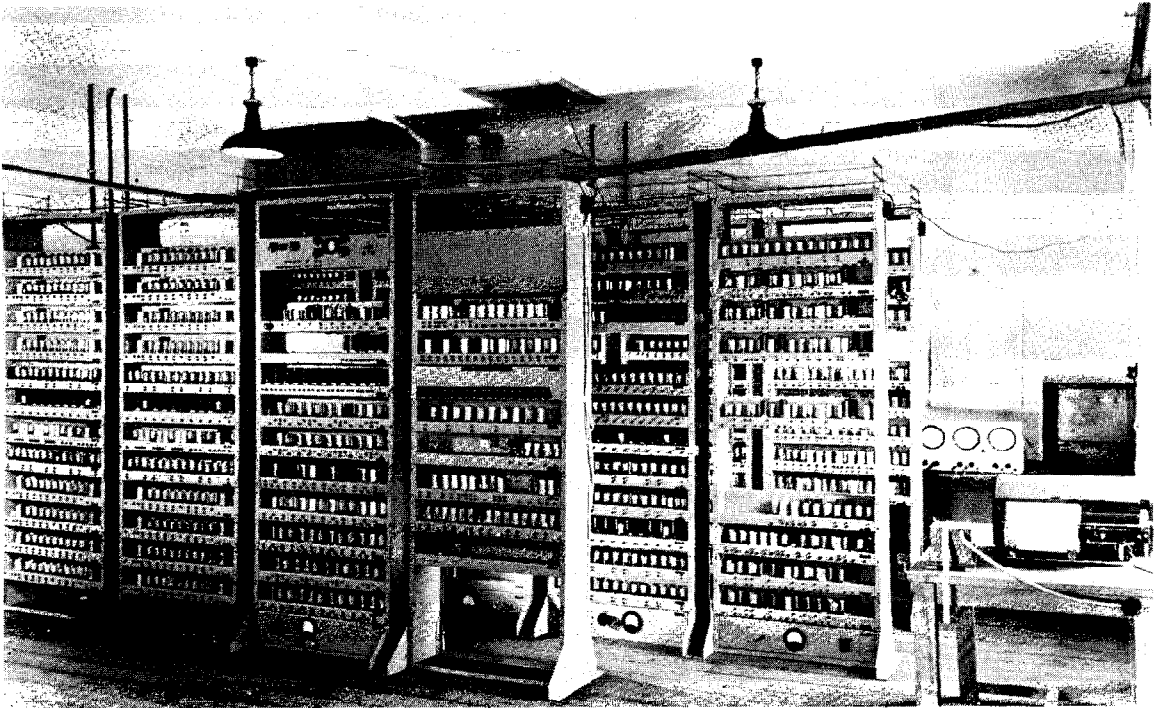


Fig. 1. The EDSAC.

for the transfer of five binary digits from the tape to the memory, and vice versa.

Operation of the machine could not start until a short standard sequence of orders, known as the "initial orders," had been transferred into the ultrasonic memory from a mechanical read-only memory formed from a set of rotary telephone switches. The space that the initial orders occupied in the memory could be re-used when they were no longer required for reading the input tape. The initial orders determined the way in which the instructions were punched on the paper tape, and this was quite an advancement for the period.

One row of holes, interpreted as a letter, indicated the function; this was followed by the address in decimal form, with leading zeros omitted and terminated by a code letter. In the first set of initial orders to be used, this code letter merely determined whether the address referred to a short or a long location; before the end of 1950, however, these initial orders had been replaced by a more elaborate set in which the terminating characters were used to provide relocation facilities for blocks of instructions or data punched on the tape.

The EDSAC did its first calculation on May 6, 1949, and ran until 1958, when it was finally

switched off,

REFERENCES

1950. Wilkes, M. V. "The EDSAC (Electronic Delay Storage Automatic Calculator)," *MTAC*, Vol. 4, p. 61.
1956. ———, *Automatic Digital Computers*. London: Methuen; New York: John Wiley.

M. V. WILKES

EDUCATION, COMPUTERS IN HIGHER. See HIGHER EDUCATION, COMPUTERS IN.

EDUCATION IN COMPUTING SCIENCE

For articles on related subjects see **COMPUTER-ASSISTED INSTRUCTION**; **COMPUTER-ASSISTED LEARNING AND TEACHING**; **COMPUTER-MANAGED INSTRUCTION**; **COMPUTER SCIENCE**; and **HIGHER EDUCATION, COMPUTERS IN**.

For articles on related terms **see** **ANALOG COMPUTERS**; **ARTIFICIAL INTELLIGENCE**; **COMPUTABILITY**; **COMPUTER GRAPHICS**; **DATA STRUCTURES**; **FORMAL LANGUAGES**; **HEURISTICS**; **HYBRID COMPUTERS**; **INFORMATION RETRIEVAL**; **INFORMATION SYSTEMS**; **NUMERICAL ANALYSIS**; **PROGRAMMING LANGUAGES**; **SEQUENTIAL MACHINES**; **STORED-PROGRAM CONCEPT**; and **SYSTEMS PROGRAMMING**.

The forerunners of the modern electronic stored-program computer were first developed at United States universities in the decade of the 1940s, mainly in response to military needs during World War II. However, a lengthy gap then ensued between the conception of computers at universities and their attendant application to the host of educational, research, and administrative processes at these universities. The beginnings of the university computer installation, or the university "computing center" organizational unit, dates only to the mid-1950s; in fact, it was only during the period from 1960 to 1965 that the "computer revolution" really took hold at United States universities. In some cases, the computing center began as a separate organization; more generally, the computing center evolved from a computer initially installed in a department of mathematics or school of engineering for research purposes. Today's computing science curricula often reflect these origins of the computing center by emphasizing the mathematical or engineering content of computing.

University Programs. The university academic program in computing began in the mid-1950s under the pressure of early users of computing equipment, or of the computing center staff deluged with questions about the use of these new devices. Initially, the "educational program" might consist only of a short, noncredit course given by the computing center staff; such a course mainly emphasized how to program a problem for computer solution (usually in machine or assembly language). At times, some of the material was absorbed into a regular course in mathematics or engineering, generally in three or four lectures. However, with the rapid growth of university computing installations during the 1960 to 1965 period, it became necessary to establish more formal educational programs in computing.

Usually, these programs started with a collaborative effort between the computing center and the department of mathematics or school of engineering,

based again on the organizational origin of the computing center. One of the most influential of these early efforts took place at the University of Michigan, and subsequently at the University of Houston, during the period from 1959 to 1962. These efforts, conducted jointly by the computing center and the college of engineering, were aimed less at establishing computing science as a distinct academic discipline and more at the "Use of Computers in Engineering Education." At approximately the same time, Stanford University, through the joint efforts of its computing center and the department of mathematics, was establishing the discipline of computing science as an optional field of study in the department of mathematics.

These early efforts were capped by the creation of separate departments of computing science. In 1962, Stanford University established a Department of Computer Science in the School of Humanities and Sciences; in the same year, Purdue University created a Department of Computer Science in the Division of Mathematical Sciences. In each case, the bond between the service and academic functions of computing was made evident by the fact that one person was both director of the computing center and chairman of the department; this pattern was followed subsequently by other universities. Another pattern established by Stanford and Purdue was that of initially offering only graduate programs in computing science, at the master's and doctorate levels. This reflected the thinking at the time that there could be no well-defined undergraduate program in computing science, and that specialization in computing should start only at the graduate level. (It also reflected the fact that there were few professors qualified to teach computing at the time.) The arguments for and against a highly concentrated program in computing at the undergraduate level continue even today, and we will return to this point later in this article.

By the mid-1960s, events in computing science education were proceeding at a dynamic pace. Government and quasi-government reports made recommendations that spurred the growth of computing science academic programs. Two of these were of particular importance. The National Academy of Sciences report on "Digital Computer Needs in Universities and Colleges" (1966) recommended among other things that campuses should "increase as rapidly as possible the number of specialists trained annually as computer specialists and the support of pioneering research into computer systems, computer languages, and specialized equipment." The President's Science Advisory Committee

EDUCATION IN COMPUTING SCIENCE

report on "Computers in Higher Education" (1967) recommended that "the Federal Government expand its support of both research and education in computer sciences." These reports helped obtain government and university support for the new discipline.

During the same time period, university-sponsored conferences produced reports and books, such as "University Education in Computing Science" (Finerman, 1968), indicating that computing science was truly an emerging academic discipline and not a short-lived curiosity item. Indeed, the "intellectual respectability" of computing science was a controversial issue in the decade of the 1960s. Many educators raised the point that the computer is just a tool, and that a body of study based upon a tool is not a proper academic discipline; others indicated that computing science is not a coherent discipline but rather a collection of bits and pieces from other disciplines; still others felt that computers were not that important and were not proper objects of academic interest. However, by and large, this skepticism was itself short-lived, at times delaying but not preventing the eventual start of academic programs in computing.

Similarly, computing, mathematics, and engineering professional societies sponsored studies of the curricula effects of the new discipline. Reports of the Mathematical Association of America and the Commission on Engineering Education recommended changes in existing academic programs to assure that students in mathematics and engineering received adequate preparation in computing. This preparation was necessitated by the fact that a growing number of mathematics and engineering majors found themselves working in the computing field soon after graduation. The studies of the Association for Computing Machinery (ACM) had the most widespread effect. ACM chartered a Curriculum Committee in Computer Science to recommend necessary academic programs. The subsequent report, "Curriculum 68" (Atchison et al., 1968), for the first time defined the scope and content of a recommended undergraduate program in computing science. Later, ACM also chartered a Curriculum Committee on Computer Education for Management. This committee issued two principal reports, one on undergraduate and the other on graduate programs in information systems. We will return to the recommendations of these committees later in this article.

The effect of all these studies, conferences, and reports was a proliferating and seemingly endless number of computing science academic programs.

These now abound at two-year colleges, four-year colleges, universities, and (more recently) have been introduced into secondary schools. Thus, courses in this subject now span the range from high schools through the doctorate. The programs go by different names, such as computing science, information science, data processing, and information systems. (The reader will have noticed that I prefer the phrase "computing science" to the more commonly accepted "computer science." This reflects a personal perspective that "computing" is the more comprehensive term of the two. However, I note that "computer science" departments are much more prevalent than "computing science" departments.) Each name denotes a somewhat different emphasis; for example, computing science indicates a mathematical and scientific flavor, while information systems indicates computing applied to organizational systems. The programs may be housed in a department of computing science, a combined department of electrical engineering and computing science, or given as an option in mathematics, engineering, or business administration. The academic program is now separate from the computing center, and rarely is the same person in immediate charge of both activities.

Nonuniversity Educational Programs.

We have noted that computing science educational programs originated at universities and spread downward, from graduate to undergraduate to two-year colleges and then to high schools. In the subsequent sections of this article we will deal almost exclusively with the university program, undergraduate and graduate. The university is where an increasing number of students are receiving their education in computing science, where the discipline of computing is being molded, and where the educational programs in computing (while still quite diverse) have their greatest cohesiveness. In this section, we note briefly other educational programs in computing, specifically those offered by high schools, manufacturers of computing equipment, and private technical schools or institutes. We also discuss the educational programs at two-year colleges, in some ways similar to the technical school and in other ways a preparation for four-year undergraduate work.

HIGH SCHOOL PROGRAMS. It is difficult to describe the high school educational program in computing, for as yet there is no such identifiable program. A growing number of high schools are offering courses in computing, and a growing number of high school graduates have received some

exposure to computing concepts. Yet, progress has been slow and there is no discipline of computing at the high school level. There are several reasons for this. First, very few high school teachers have taken computing courses, and still fewer understand the subject well enough to teach it. Second, to date, computing equipment has been relatively expensive, and not many high schools are affluent enough to install their own computers, or even terminals to nearby computing centers. Third, there is some doubt as to whether the discipline of computing is fundamental enough-as, for example, are English, mathematics, or physics-to teach at the high school level.

Changes are taking place. A growing number of teachers are being exposed to computing; computing equipment is becoming more affordable; attitudes about computing are being modified as the subject becomes better understood. In general, however, computing programs at the high school level, where they exist at all, consist of a cursory introduction to programming, given either as part of a mathematics course or as a separate course. This situation probably will not change materially in the immediate future.

MANUFACTURERS' PROGRAMS. Manufacturers have been offering courses in computing for years. Indeed, this informal educational activity predates the formal university educational program; probably the greatest number of people entering the computing industry have been exposed only to manufacturer-offered courses. Most such courses are intended only for customers of the manufacturers, and most are concerned only with the equipment offered by those manufacturers. In some cases, courses are as much an exercise in marketing as they are in education; the object is to sell the doubtful customer on the need for bigger and better computers, especially those offered by the manufacturer giving the course. However, some manufacturers have attempted to maintain high standards; some have separated the educational program from the marketing activity. Yet, on the whole, performance has been rather spotty. For years these courses were given free to customers. More recently, manufacturers have begun to offer certain courses for a fee. In these cases, the manufacturer-offered educational program may rival that of the private "computing institute."

TECHNICAL SCHOOL PROGRAMS. Private schools for training technicians have been operating for years. In many fields they serve a worthwhile function by preparing people for jobs as secretaries, dental technicians, draftsmen, and the like. When the computing industry started expanding rapidly, a

large number of private industries began offering educational programs in computing. There are many jobs in the industry for which technician training is worthwhile, and the technical school graduate should be qualified to assume such jobs.

Unfortunately, at times, the computing institute may intimate that its training will prepare students for well-paying professional jobs in the computing industry. The technical school graduate often discovers too late that most such positions are filled by college graduates, that the professional career path in computing, as in most other fields, requires a college education. People trained as secretaries are well aware that they will not be hired for, or can advance to, executive positions; draftsmen know that they will seldom become professional engineers-at least not with technical school training only. Perhaps because of the newness of the field and the attendant absence of uniform standards and professional certification, this same fact is not as yet well recognized in computing. Currently, efforts are under way by computing societies to develop certification examinations for computing practitioners, and a number of states have given serious attention to the need for certification. These efforts may, in time, lead to a sharper and more accepted distinction between technicians and professionals.

COMMUNITY COLLEGE PROGRAMS. Two-year community (or junior) colleges have grown phenomenally in recent years, both in quantity and in scope of offerings. Twenty years ago, the community college was rather rare, and usually specialized in such areas as agriculture, forestry, and mining. Today, the community college has become as broadly based and diversified as its university cousin. The community college serves a twofold purpose. One is to train the student for a position as a technician. For these graduates, the two-year Associate degree is proof of better standards than those usually maintained by the technical school; the degree is also proof of a more well-rounded education.

The second purpose of the community college is to serve as a bridge between the high school and the four-year college or university, especially for those students uncertain of their desire or ability to continue with higher education. For these people, the Associate degree may be an intermediate step on the way towards a bachelor's degree.

Two-year colleges have rapidly expanded their educational programs in computing to fill the same two needs. For those students wishing to terminate at the technician level, the community college education is a more satisfying alternative than the private computing institute. For others, the com-

EDUCATION IN COMPUTING SCIENCE

munity college is a valuable stepping stone to the university computing science program. But there are some problems associated with this educational background.

Students terminating after two years and entering industry suffer the same identity problem as the technical school graduate. Indeed, they are more than technicians, but not quite the same as college graduates. More often than not, the career paths open to them are technician-oriented. On the other hand, graduates wishing to continue toward the bachelor's degree sometimes find the transition quite difficult. Community college standards are not always the same as university standards; community college courses are not always identical or even similar to the corresponding courses at the university.

Some of these difficulties are being addressed. Community colleges and universities have been cooperating in facilitating the transition by making courses more compatible, although it still remains a problem. Moreover, the technician versus professional point is far from reconciled. Increasingly, as the "computing profession" evolves and becomes better defined, the broader educational scope of a bachelor's degree becomes a prerequisite for a professional career.

We will not separately detail the usual curricula at two-year colleges. In some cases, these are similar to freshman and sophomore level computing courses at universities. In other cases, the differences are more visible. By and large, university programs are "scientifically" oriented, emphasizing both the scientific underpinnings of computing and the scientific or engineering applications. Two-year college programs tend to emphasize the practical aspects and the business applications (for example, accounting) of computing. The four-year university program allows more time to take unrelated courses outside those in computing, mathematics, and associated technical disciplines. (This is not always so, as we will discuss later.) Community college programs, because of the shorter time span, are more intensely oriented to courses in computing, business mathematics, accounting, and other technical areas.

Trends in Undergraduate Education. Let us now consider some general trends in undergraduate education, particularly in the science, technology, and professional fields, and the relation of these trends to academic programs in computing science. At the present time, the general practice in the United States is that technology-oriented students receive an undergraduate education that is

highly specialized in their particular fields. For example, engineering students take most of their courses in engineering and related science (mathematics, physics, and chemistry) fields. This practice is followed even more in other countries, where students may take all courses in their particular faculty. In the past few years, various institutions have begun to question whether or not this practice of specialization at the undergraduate level of study denies the student the benefits of a more general education. For example, the Massachusetts Institute of Technology several years ago established a Commission on Undergraduate Education to study the academic program and recommend necessary changes. The Commission identified three basic aspects of undergraduate education and some shortcomings of excessive specialization.

The first aspect concerns integration of knowledge. Modern problems require that students develop the ability to synthesize as well as analyze; these problems point to the need for interdisciplinary curricula. The second deals with facts and values. The most difficult problems we face are those that relate facts and values; these require that intellectual tools from the humanities be included in the programs of engineers and scientists. The third concerns education for citizenship in a democracy. To some people these days, this may sound somewhat old-fashioned, but it has never been more important for students to understand the nature of a democracy and their roles as individuals in it.

This argument of increasing specialization versus a more general education at the undergraduate level is also a controversial issue in computing science education. On the one hand, there are those who believe that the computing science program must be highly specialized, and must emphasize computing, mathematics, and related technical subjects. Otherwise, the computing science major will not be prepared to take a job in industry or continue with his graduate education. On the other hand, there are those who believe that a more liberal undergraduate program is especially necessary in computing. Computing pervades many disciplines, and the computing student should have a background in those disciplines; further, if computing technology is to contribute to the meaningful development of society, the computing scientist must be a well-educated and informed citizen.

There are many more such arguments pro and con, and the controversy continues unabated. At the present time, most university curricula in computing science are highly specialized, following the general trend in undergraduate professional education.

The Undergraduate Curriculum. Because computing science is so new, there is as yet no well-established standard curriculum in this discipline. The undergraduate program varies from university to university, depending upon such things as the resources available, the amount of specialization deemed useful, and the interests of the faculty. Even the content of specific courses is, in some cases, quite variable. As we noted earlier in this article, the most scholarly attempt made to date in defining the scope and content of an undergraduate program in computing science has been the work of the ACM Curriculum Committee, "Curriculum 68" (Atchison et al., 1968). Indeed, this report has had a most profound effect on shaping the direction of computing education. Many institutions view its recommendations as the definitive yardstick by which to measure the adequacy of their programs.

Curriculum 68 contains detailed information on four beginning courses (prefixed by the letter B), nine intermediate courses (I), and nine advanced courses (A), a total of 22 undergraduate courses. We list these courses together with a brief description of each, not so much to specify detailed content as to indicate the general range and breadth covered by computing science.

B1. *Introduction to Computing*

The basic knowledge of algorithms, languages, programming, and program structure necessary to use computers effectively.

B2. *Computers and Programming*

The basic structure, language, and internal behavior of computers and the relation among these elements.

B3. *Introduction to Discrete Structures*

Fundamental algebraic, logical, and combinatoric concepts from mathematics and their application to computing science.

B4. *Numerical Calculus*

Fundamental numerical algorithms, in such areas as linear and nonlinear equations, used in scientific work.

I1. *Data Structures*

Elements of data involved in problems, structure of storage media, methods of representing, and techniques for operating on structured data.

12. *Programming Languages*

Specification of syntax and semantics as applied to algorithms, list processing, string manipulation, and simulation languages.

13. *Computer Organization*

A continuation of concepts introduced in course B2, the organization, logic design, and components of digital computers.

14. *Systems Programming*

Software organization and the role of data structures and programming languages in the design and organization of computing systems.

15. *Compiler Construction*

The organization of compilers, including symbol tables, lexical scan, syntax scan, and object code generation.

16. *Switching Theory*

Theoretical principles and mathematical techniques involved in the design of digital systems logic.

17. *Sequential Machines*

Definition and representation of finite state automata and sequential machines, and decision problems of finite automata.

18. *Numerical Analysis I*, and

19. *Numerical Analysis 2*

Mathematically rigorous and computer-oriented methods in the solution of equations, linear systems, and differential equations.

A1. *Formal Languages and Syntactic Analysis*
Theory of context-free grammars and formal languages, and syntactic recognition techniques.

A2. *Advanced Computer Organization*

System design problems and comparison of specific examples of solutions for various computer organizations.

A3. *Analog and Hybrid Computing*

Analog, hybrid, and related digital techniques, operational characteristics of analog components, and conversion methods.

A4. *Systems Simulation*

Simulation and modeling of discrete systems, simulation methodology, and design of simulation experiments.

A5. *Information Organization and Retrieval*

Natural language processing, particularly as applied to the design and operation of automatic information systems.

A6. *Computer Graphics*

Problems and techniques for handling graphic information, such as line drawings, block diagrams, and handwriting.

A7. *Theory of Computability*

Use of abstract machines and models in the study of computability and computa-

tional complexity.

A8. Large-Scale Information Processing systems

The design, organization, and integration of hardware, software, procedures, and techniques.

A9. Artificial Intelligence and Heuristic Programming

Application of computing systems to problems that attempt to achieve goals normally considered to require human mental capabilities.

Curriculum 68 recommends that the major in computing science should consist of ten required computing courses (B1 to B4, 11 to 14, and two from 15 to 19), plus perhaps three elective computing courses. In addition, the report lists eight supporting courses in mathematics, covering such areas as calculus, linear algebra, algebraic structures, and probability and statistics. The report notes that an academic program in computing science must be well based in mathematics, since computing science draws so heavily upon mathematical ideas and methods. Consequently, at least six of the mathematical courses listed should be required, and additional electives in mathematics should be encouraged. The program prescribed in Curriculum 68 reflects the viewpoint of those advocating a strong specialization in computing at the undergraduate level; as such, it follows the traditional pattern of most scientific and engineering undergraduate programs. The large component of computing and mathematics courses recommended (between one-half and two-thirds of the total undergraduate course load) plus technical electives in computer-related disciplines, leaves little room for the nontechnical subjects in the humanities and the social sciences.

More liberal undergraduate programs would perhaps limit the recommended number of computing and mathematics courses to one-third of the total course load, and would require that another one-third be taken in the humanities and the social sciences. By allowing flexibility in the remaining one-third, such programs would permit students to take a double major-in computing and in some other discipline such as physics, psychology, or economics.

BUSINESS DATA PROCESSING. While Curriculum 68 emphasized the mathematical and scientific content of computing science education, it paid little attention to the business-oriented computing student. Computers were originally developed for the scientific and engineering applications. Somewhat

later, however, the use of computing systems for commercial applications, the "business data processing" aspect, began to exceed the scientific applications. Today, more computers are applied to, and more computing practitioners are engaged in, the data processing end than in the scientific. Yet students interested in this area find few academic programs meeting their needs, except for the host of "data processing" programs at community colleges or sometimes in the undergraduate and graduate programs in schools of business.

But perhaps a more subtle shortcoming of the scientifically oriented computing program is that it does little to prepare students adequately for the more recent applications of computing technology. As users have gained experience and computing capabilities have assumed a more general-purpose nature, the distinction between "scientific" and "business" applications has become increasingly blurred. Many modern applications require the successful integration of the systems analysis approach associated with data processing and the more rigorous mathematical analysis approach associated with scientific processing. Applications in space technology, library information retrieval, airline reservations, and banking are but a few examples of these "large scale" systems.

The large systems bring together many organizational units, technical disciplines, and people-oriented procedures; information flows through and across organizations, and causes problems at the interface; computer programs written by different organizations must be tested, accepted, and integrated into one operational system. Computing science academic programs produce students expert in computing technology but not in management organization, human and organizational behavior, systems analysis and design, economics, and the like. Yet, it is just this type of knowledge as much as the mastery of computing technology that is required to properly implement many large-scale applications.

MANAGEMENT INFORMATION SYSTEMS. As indicated earlier in this article, several years ago the ACM chartered a Curriculum Committee on Computer Education for Management. The approach of this committee has been to design curricula in the field of "information systems," systems in which computing technology is applied to the information needs of the organization. This concept is evident in the two major reports presented by Ashenurst (1972) and Couger (1973). Couger's report on the undergraduate program recommends a curriculum that gives students knowledge of the organization, its information needs, and its behavior, as well as of

computing technology. Students enrolled in this program are required to take certain prerequisite and co-requisite introductory courses in economics, psychology, mathematics, statistics, and computing. (The computing prerequisite may be quite similar to the "Introduction to Computing" course detailed in Curriculum 68.)

Let us examine these courses briefly. They are in four groups: background (prefixed by UB), computing (UC), analysis of organizations (UA), and development of systems (UD).

- UB1. *Operational Analysis and Modeling*
Analytical and simulation modeling techniques useful in decision making in the system design environment.
- UB2. *Human and Organizational Behavior*
Principles governing human behavior, particularly in organizations, and use of computer-based information systems in organizations.
- UC1. *Information Structures*
Structures for representing the logical relationship between elements of information, and techniques for operating on such structures.
- uc 2. *Computer Systems*
A working view of hardware and software configurations considered as integrated systems,
- uc 3. *File and Communication Systems*
Basic functions of file and communications systems, current realization of these systems, and analysis of these realizations.
- uc 4. *Software Designs*
Complex programming tasks; and subdividing such programs for maximum clarity, efficiency, and ease of maintenance and modification.
- UA8. *Systems Concepts and Implications*
The basic concepts involved in the systems point of view, the organization as a system, information flows, and information systems.
- UD8. *Information Systems Analysis*
Analysis of the design of an information system intended to facilitate decision making and planning and control.
- UD9. *System Design and Implementation*
The knowledge and tools necessary to develop a physical design and an operational system from the logical design.

Not all these courses are intended to be required

for all students. In particular, students may decide to take an organizational or a technological concentration. For the former, there are two courses, UC8 and UC9, in place of the four courses UC1 to UC4; for the latter, course U132 may be eliminated. The report also stresses the concept of a double major in information systems and (for example) accounting or engineering. It gives illustrations of sample programs for these two double majors. It is still uncertain whether such programs would be housed in a school of business, a school of engineering, department of computing science, or indeed whether universities would give such programs the priority given in the past to the scientifically oriented programs in computing. It may be that in future years the effects of this report on undergraduate programs in information systems will be as widespread as those of Curriculum 68.

In any event, most departments of computing science or information systems are generally oriented more toward "software" than "hardware." By necessity, and with some exception, students interested in the hardware aspects of computing normally major in departments of electrical engineering. (Indeed, graduates of such programs are referred to as hardware "engineers" rather than computing "scientists," although the distinction is not meaningful.) Courses at the interface (e.g., switching and microprogramming) may be given in computing science or electrical engineering, or both. As noted earlier, computing at some universities is an option in electrical engineering; at others there are combined departments of electrical engineering and computing science.

As a final note on undergraduate programs in computing, many undergraduate courses in computing attract not only the major in computing science (or information systems) but also students majoring in other disciplines. The President's Science Advisory Committee report (Pierce et al., 1967) estimated that about three of every four college undergraduates have need for some educational exposure to computing. The extent of such exposure obviously depends upon the fields in which students are majoring. Those in the liberal arts might benefit only from some introductory knowledge of computing technology. Students majoring in the sciences, engineering, or business administration require a great deal more, in some cases extending to a "minor" or double major in computing. Usually, no special "service" courses in computing are intended mainly for the noncomputing science major. However, the computing science program performs a necessary service function for these students.

EDUCATION IN COMPUTING SCIENCE

The Graduate Program. Although graduate programs in computing predate the undergraduate, there are few descriptions in the formal literature of graduate curricula. Curriculum 68 contains a brief description of the master's and doctoral program. Finerman's (1968) book is more explicit, since it derived from a conference on graduate academic programs.

A master's program in computing science may be terminal or nonterminal. In the first case, the goal is to develop programming and applied problem-solving capabilities so that the graduate can qualify for a senior position in industry or research organizations. The goal of the nonterminal program is to develop the intellectual capabilities so that the graduate can continue to the doctoral program (although some students go directly from the bachelor's degree to the doctorate). This approach is in the pattern of the traditional master's programs for so-called scientific and professional disciplines. In many sciences, the master's program is viewed as the testing ground for further graduate study. Students who prove they have the necessary capabilities can continue for the doctorate; those who cannot, terminate with the master's degree. However, in professional disciplines? the terminal master's degree is an honorable goal, since most people have no need for the doctorate. In business administration, for example, several excellent universities offer the terminal master's program. Computing science, both a scientific and a professional discipline, increasingly differs the two choices, depending upon the inclination of the student.

The master's program generally has three areas of specialization. The first encompasses information structures and processes. This is quite abstract and theoretical, involving advanced mathematical concepts; it includes such topics as computability, formal languages, and switching and automata theory. The second category covers information processing systems. This deals with practical techniques in such areas as computer organization and design, programming languages, operating systems, and assemblers and compilers. The third category comprises methodologies. This deals with a broad spectrum of techniques appropriate to various computing applications, such as numerical analysis, text processing and graphics, symbol manipulation, simulation, information retrieval, and artificial intelligence.

The master's program in the last two categories -information processing systems and methodologies-might be terminal or not; at many institutions it is assumed that students in these categories

will terminate. Students in these areas take one or more courses in the theoretical aspects of computing science, but most courses are in their particular field of specialization, drawn from the intermediate and advanced courses in Curriculum 68 or from graduate versions of these courses. The theoretically inclined student is usually interested in continuing toward the doctorate. Therefore, the first category-information structures and processing-is generally a **non-terminal** program; again, students in this category are required to take some courses in the other areas. At many institutions, the terminal master's student may elect to perform thesis work, or not; if not, two more courses may be substituted for the thesis or a project may be required. Nonterminal students are expected or required to submit a thesis as further proof of their qualification for the doctorate. Many students enter the master's program with undergraduate degrees that are not in computing science but in the sciences or engineering. Thus they have the essential of a good background in mathematics, especially important for those who expect to continue past the master's degree.

In contrast to this approach to computing science, Ashenhurst (1972) considers the professionally oriented master's program. The philosophy expressed in the undergraduate curriculum carries forward to the graduate level. The program in information systems deals with computing technology as applied to the organization and its systems. As in the undergraduate program, there are four groups of courses. The first involves two background courses: one in operations research and the other in behavior in organizations. The second group, dealing with the environment in which systems function, involves four courses in the organization and systems requirements. The third group contains four courses dealing with computing and information technology. Finally, the fourth group integrates the previous two groups with three additional courses in the analysis and development of systems. The courses are more advanced versions of those at the undergraduate level. Students can enter this program with a background in computing, economics, or other business areas, although the program is equally useful for those with an engineering background. As we discussed previously, computing technology is applied to large engineering or scientific systems as well as to business or commercial systems; in many applications, the "scientific" or "business" distinction is meaningless. The entering student is required to have prerequisite undergraduate courses in **mathematics**, statistics, computer programming, **economics**, and psychology.

Most doctoral programs are for students with theoretical or research interests. Courses at this level especially reflect the special interests of the faculty members. In general, however, they are similar to the master's degree programs in the following areas: logical design, switching theory, computer circuits and devices; computer organization, programming languages, compiler and operating systems; computability, formal languages, automata; numerical mathematics, operations research; methodologies such as artificial intelligence, simulation, and modeling. The doctoral thesis, drawn from one or more of these areas, lies at the heart of the doctoral program. It is the means by which the student demonstrates the capability for original contribution to knowledge. This is, of course, the fundamental requirement for the doctorate;

Some universities are reexamining academic programs at the graduate as well as at the undergraduate level. In some cases this examination may lead to broader (and longer) interdisciplinary master's degree programs; in others, predoctoral intermediate degree programs may be established, which do not require original contribution to knowledge but recognize a dimension of excellence in another area. In general, however, the specialized master's degree and the doctorate still are regarded as the accepted graduate programs.

Summary. Formal education in computing science is quite new, dating back only to the early 1960s. Educational programs originated at universities, resulting from the increasing use of computers to service the needs of university students, faculty, and administrators. Today, most colleges and universities offer academic programs in computing, mainly as a separate discipline but sometimes as an option in a related discipline. As can be expected in such a new field, the educational program still has fuzzy edges; at times, it overlaps applied mathematics, electrical engineering, business administration, and other disciplines. Yet, in just a few short years, it has become a visible and influential area of study.

Computing science undergraduate programs also provide a service function by making courses available to the major in other disciplines. Usually, these students require some computing courses so that they can better apply computing methods to their fields; often, however, these students become computing practitioners after graduation. Perhaps because of the newness of the field, industrial organizations fill their computing positions with graduates from many disciplines in addition to

computing.

Although the discussions in this article apply primarily to computing education in the United States and Canada, experiences in other countries are quite similar. The major difference is that computing science educational programs in other countries were introduced later than those in North America. For example, with some exceptions, western European and Israeli universities initiated such programs around the late 1960s, South American universities around 1970, and African universities during the early 1970s.

One result of these educational programs is a recognition that computing science involves more than the study of a tool. The computer has given entirely new scope to the whole spectrum of computation applications, so much so that people who work with computers have found new approaches to problem solving, even in areas to which the computer as a tool may never be applied. Too often our early educational programs emphasized the study of the computer as a tool; many programs, especially those below the college level, still do. There is now, however, an increasing awareness that the use of the computer stimulates and modifies intellectual processes, and as a result makes it possible for man to expand his intellectual capabilities. This added dimension must be part of any academic program in computing science or information systems.

REFERENCES

The early efforts to bring computing methods into engineering education are described in three related volumes:

1960. University of Michigan Study, "Electronic Computers in Engineering Education." Ann Arbor: University of Michigan.
1961. University of Michigan Study. "Use of Computers in Engineering Education, Second Annual Report." Ann Arbor: University of Michigan.
1962. University of Houston Study. "Use of Computers in Engineering Education-A Report of the Advanced Science Seminar." Houston, Texas: University of Houston.

There were two principal government-sponsored studies on computing in universities during the mid- 1960s. Both gave background information on the use of computers in universities and recommended government financial support for computing education:

EDVAC

1966. Rosser, J. B., et al. "Digital Computer Needs in Universities and Colleges." Washington, National Academy of Sciences/National Research Council.
1967. Pierce, J., et al. "Computers in Higher Education," The President's Science Advisory Committee, The White House. Washington, DC.: U.S. Government Printing Office.

There are four principal references for undergraduate and graduate programs in computing. Three have been published in the monthly periodical, *Communications of the Association for Computing Machinery* (CACM), and the fourth is a monograph of the ACM:

1968. Atchison, W., et al. "Curriculum 68," A Report of the ACM Curriculum Committee on Computer Science Education, *CACM*, Vol. 11 (March), pp. 15 1-197.
1972. Ashenhurst, R. (Ed.). "Curriculum Recommendations for Graduate Professional Programs in Information Systems," A Report of the ACM Curriculum Committee on Computer Education for Management, *CACM*, Vol. 15 (May), pp. 363-398.
1973. Couger, J. D. (Ed.). "Curriculum Recommendations for Undergraduate Programs in Information Systems," A Report of the ACM Curriculum Committee on Computer Education for Management, *CACM*, Vol. 16 (December), pp. 727-749.
1968. Finerman, A. (Ed.). "University Education in Computing Science," ACM Monograph. New York: Academic Press.

A. FINERMAN

EDVAC

For articles on related subjects see **DIGITAL COMPUTERS**: Early; ENIAC; **STORED-PROGRAM CONCEPT**; and **VON NEUMANN MACHINE**.

For articles with related biographical information see **ECKERT, J. PRESPER**; **MAUCHLY, JOHN W.**; and **VON NEUMANN, JOHN**.

The EDVAC (Electronic Discrete Variable Automatic Computer) was a direct outgrowth of the work on the ENIAC. During the design and construction of the ENIAC in 1944 and 1945, the need for more storage (only twenty lo-decimal digit numbers in the ENIAC) was realized. The experience with acoustic delay lines for radar range measurement led to the concept of recirculating storage of digital information. The group at the Moore School of Electrical Engineering at the University of Pennsylvania started developmental work on mercury delay lines for such storage, and initiated the design of the EDVAC.

This was the first stored-program computer; the instructions controlling the computational process are stored in the same way that data is stored. The basic logical ideas were described by von Neumann (1945), and computers based on such designs have come to be known as "von Neumann computers." The principles involved in the EDVAC design exerted a strong influence on the computers that followed it.

The EDVAC had about 4,000 tubes and 10,000 crystal diodes. It used a 1,024-word recirculating mercury-delay line memory, consisting of 23 lines, each 384 μ s long. The words were 44 bits long. Instructions were of the four-address type (4-bit operation code and four 10-bit addresses). The arithmetic unit did both fixed and floating-point operations. Input and output were via punched paper tape and IBM cards. Information was all handled as serial pulse trains and the clock frequency was 1 MHz.

Although the conceptual design of the EDVAC was complete in 1946, and it was delivered in 1949 to the Ballistic Research Laboratories at Aberdeen, Maryland, by 1950 the entire calculator had not yet worked as a unit and was still undergoing extensive tests (Stifler, 1950, pp. 200-201). The delay in completion of the EDVAC was primarily due to the efflux of computer people from the Moore School in 1946. Eckert and Mauchly resigned and launched a commercial venture (UNIVAC). Herman Goldstine and Arthur Burks went to Princeton to work with von Neumann, and the author left to work with Turing in England. T. K. Sharpless was put in charge, but he, too, left later to go into business for himself.

The EDVAC finally became operational as a unit in 1951. An Aberdeen Proving Ground report states that during 1952, the EDVAC "began to operate on a production basis." For nine months of 1952 the average available time per week was 47.4 hours (23.3 for code checking and 24.1 for production), and the average "engineering" time was 104.8 hours. Approximately 70.4 hours of this was unscheduled maintenance; 10,000 defective tubes (over

twice the complement) and about 3,000 (of 10,000) germanium diodes had been replaced. In a later Aberdeen report, Weik notes that during 1956, the average error-free running period was approximately 8 hours, and that out of a run time of 8,728 hours, 6,752 were good (78%). This gave approximately 130 hours of "good time" per week. The EDVAC was used until December 1962 (Knuth, 1970, p. 259).

REFERENCES

1945. von Neumann, John, "First Draft of a Report on the EDVAC," Contract No. W-670-ORD-4926, U.S. Army Ordnance Department. Philadelphia: University of Pennsylvania, Moore School of Electrical Engineering, June 30.
1950. Stiffler, W. W., Jr. (Ed.). *High Speed Computing Devices*. New York: McGraw-Hill.
1970. Knuth, Donald E. "Von Neumann's First Computer Program," *Computing Surveys*, Vol. 2, No. 4 (December), pp. 247-260.

H. D. HUSKEY

ELECTRONIC CALCULATOR. *See*
CALCULATOR, ELECTRONIC.

EMULATION

For articles on related subjects see **HOST SYSTEM**; **MICROPROGRAMMING**; and **SIMULATION**.

For article on related terms see **IBM 360-370 SERIES**.

There is little agreement as to which of the many meanings attributed to emulation is valid. Rather than attempt to adjudicate the issue here, a selection of different definitions is presented in chronological sequence followed by a short discussion.

1. "Emulation is the name given to the technique introduced in the IBM System/360 machine series for aiding in the conversion problem [to System/360 from previous IBM computers in the

1400 and 7000 series]. An emulator is a package that includes both special hardware and a complementary set of software . . . which runs in the manner of an interpretive routine simulator program but is 5 or even 10 times as fast as a purely software simulator." (S. G. Tucker, 1965)

2. "[An] emulator [is] a complete set of microprograms which, when embedded in a control store, define a machine." (R. F. Rosin, 1969)

3. "Here, emulation is defined as the ability of one system to execute machine language programs written for another system." (S. Husson, p. 15, 1970)

4. "Emulation is defined here as a combined hardware-software approach to simulation. . . . An emulator is basically an extension of the host machine's architecture, hardware and software to include the range of the target machine [i.e., the machine being emulated]." (S. Husson, 1970, p. 90)

5. "Emulation (as defined by IBM) means the ability of one [stored-program digital computer] to interpret another's program *at a reasonable performance level*." (Bell and Newell, 1971)

6. "[To emulate is] to imitate one system with another such that the imitating system accepts the same data, executes the same program, and achieves the same results as the imitated system." (Clason, 1971)

Three observations can be made with respect to this list of definitions. First, as can be seen in the preceding definitions, use of the term "emulation" has been subject to the same evolutionary pressures that impact almost all computer science. Indeed, the author of this article is far more in agreement with definitions 3 and 6 than he is with his own outdated interpretation offered in 1969, and included here as definition 2.

Second, the influence of the IBM Corporation, although probably not a result of corporate policy, can be clearly seen in that definitions 1, 4, and 5 are directly or indirectly based on the original use of the term in connection with System/360. Although it might be suggested that alternatives should be found when referring to concepts other than those introduced by IBM, recent publications from that company indicate a strong trend toward a more general interpretation, such as that given in item 6.

Finally, emulation has been strongly associated with microprogramming, as seen in definition 2 and in the title of the source of definitions 3 and 4. However, similar to the trend away from the original, narrow IBM definition, the relationship between microprogramming and emulation is considered quite tenuous in many circles today.

ENGINEERING APPLICATIONS

REFERENCES

1965. Tucker, S. G. "Emulation of Large Systems," *Communications of ACM*, Vol. 8, No. 12 (December), pp. 753-761.
1969. Rosin, R. F. "Contemporary Concepts of Microprogramming and Emulation," *Computing Surveys*, Vol. 1, No. 4 (December), pp. 197-212.
1970. Husson, S. *Microprogramming: Practices and Principles*. Englewood Cliffs, NJ.: Prentice-Hall.
1971. Bell, C. G., and A. Newell. *Computer Structures*. New York: McGraw-Hill.
1971. Clason, W. E. *Dictionary of Computers, Automatic Control, and Data Processing*. Amsterdam: Elsevier.

R. F. ROSIN

ENGINEERING APPLICATIONS

For articles on related subjects see **COMPUTER-AIDED DESIGN; CONTROL APPLICATIONS; FINITE ELEMENT METHOD; NUMERICAL ANALYSIS; PERT/CPM, PROBLEM-ORIENTED LANGUAGES; and SIMULATION.**

Electronic computers have triggered a revolution in the various engineering disciplines. They cover so wide a field that it seems impossible to deal here with their full extent. We will, however, attempt to discuss the following categories:

1. Numerical and process control.
2. Engineering analysis.
3. Simulation.
4. Optimization.
5. Engineering design.

Numerical Control. One of the computer applications in production is the numerical control for machining metal parts. These parts are produced by either milling or routing. Essentially, the cutting tool is moved in a predetermined path so that a part is machined from a sheet metal or other heavier metal stock.

If, for example, we wish to cut the two-dimensional contoured shape shown in Fig. 1, the basic operation involved is a movement of the cutting tool in the x and y directions. The cutting tool is advanced in, say, the x direction when the x -

direction control receives a pulse. These pulses direct the center of the cutter, but the cutting is actually performed by the edge of the cutter. Therefore, the offset of the cutter must be determined first.

As far as a production engineer is concerned, cutting a contoured shape involves the following typical steps:

1. Translate the required part on the blueprint into a set of instructions in a user-oriented language, such as APT (Automatically Programmed Tools).
2. Describe the offset path for the cutter.
3. Produce a punched tape by the computer.

The tape, containing the information of detailed motions of the cutting tools, is used to control the milling machine.

In the three-dimensional case, the surfaces used to define the end product are of two types. One is the classic type, such as planes, spheres, and cylinders; and the other is commonly known as a "sculptured surface"--complex doubly curved surfaces. The art of machining these complex surfaces has been recently enhanced by extensions to the APT language and its numerical control program.

PROCESS CONTROL. The numerical control described above provides automation of the discrete operations. By contrast, computers may be used to provide automation of continuous operations, known as "process control." A typical example is that of a chemical plant where a definitive sequence of decisions is required to process raw materials. Once all alternative possibilities are predetermined, computers can readily be used to control the entire processing.

The purpose of automatic control of chemical plants is twofold: to achieve optimum production and to obtain the highest possible quality of product. One unique feature of a typical chemical process is the large number of parameters involved, such as flow rate, viscosity, pressure, and temperature. Non-linear relationships are used as a rule. Typically, information from various measuring stations is transmitted to a computer, which will computerize or make decisions for the control mechanisms so that an optimum operation is maintained.

Computer control is also extensively used in many other fields of engineering. Applications include automatic testing of circuit board assemblies and complex shipboard control systems.

Engineering Analysis. Many problems in engineering frequently reduce to one or two standard mathematical problems. Sometimes the explicit

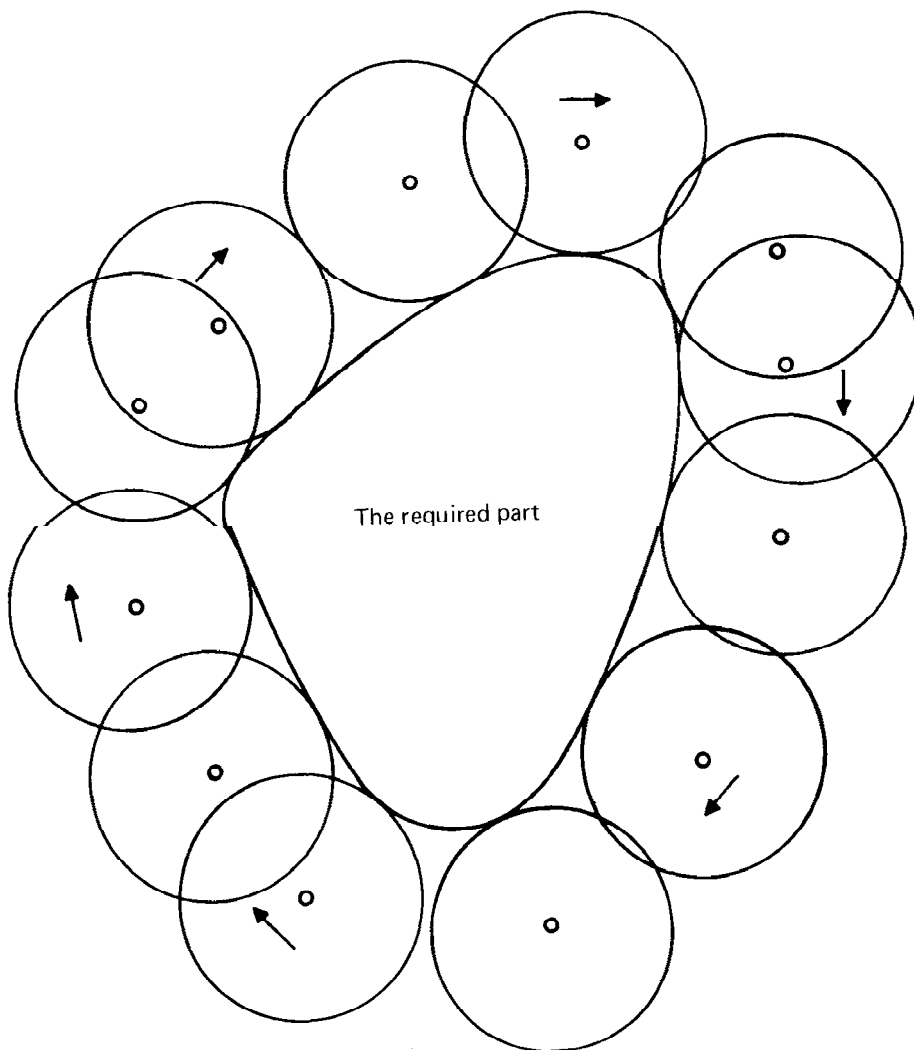


Fig. 1. Offset path for the cutter tool in numerical control. The circles indicate some of the successive positions of the cutter. The center of each circle represents the cutter center, which is directed by the control pulses.

solutions may be found, and these are easily programmed for a digital computer. More frequently, they require approximate numerical treatment.

The two most common classes of mathematical problems seem to occur in matrix manipulations and ordinary differential equations. Matrix problems may arise in structural or network analysis, vibrations, and buckling. Usually a continuous system is idealized as a discrete system, a process known as "physical discretization." Matrix problems may also arise from a large class of partial differential equations through the use of finite differences, a process often known as "mathematical discretization." The

advent of computer technology has enabled engineers to deal with matrices of very large size. Problems involving matrices on the order 100,000 have been handled successfully.

Ordinary differential equations occur in chemical reaction systems, spring-mass systems, temperature distribution, and many-body systems.

Other standard problems include the solutions of polynomial and transcendental equations, interpolation, curve fitting, and quadrature.

Since the advent of computers, the *finite element method* has gained popularity in various fields of engineering, including structural mechanics, heat

ENGINEERING APPLICATIONS

transfer, fluid mechanics, and soil mechanics. In this method, the two- or three-dimensional domain of interest is first divided in small geometrical elements—a physical discretization. The elements may be of triangular, quadrilateral, tetrahedral, hexahedral, or other well-defined forms. The accuracy of the results is readily increased when the region is divided into smaller elements. The effect of the geometrical form of the elements on the behavior of the associated matrices has been extensively studied.

Simulation. An engineering system or device may be represented by a mathematical model. Such modeling, or simulation, is often used to aid in the engineering design. One chief advantage of simulation is the possible elimination of the actual building of expensive engineering systems before testing. A typical example is chemical process and control simulation. By means of modeling, the process equipment and controllers are literally built within the computer. (The equipment may be a heat exchanger or a chemical reactor, while the controller may include valves and feed devices.) Once the information enters the computer to represent the factors for the real process, the process itself is dynamically simulated. During this simulated process, various revisions may be made and disturbances introduced. A new control element, say, may be inserted, or the setting of a control element may be changed.

Optimization. This type of application is concerned mainly with the operational and system aspects of engineering problems. For example, we deal with how to obtain the largest possible number of gusset plates from a sheet of rolled plates, rather than how to design gussets. In a large system, the field data and measurements are usually integrated with some mathematical models and are processed on a large-scale computer. Important techniques include linear and dynamic programming and network analysis. PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method) are also used to assist engineering decision making.

Engineering Design. Without computers, for 'most design problems the engineer would either make a "guestimate" from his experience or, at best, estimate only one or two alternatives in his design. When a digital computer is used, a complete comparative study can be readily obtained to show the effect of numerous parameters. This is a typical situation where a computer is used to perform bulky and repetitive calculations.

An example is the structural design of orthotropic steel highway bridges. This type of bridge is found to require less steel than a bridge of the conventional type (plate girders) for medium or long spans. For a long time the design of orthotropic bridges was not popular, probably because it is complicated and time consuming. Now it seems almost routine to accomplish several types of bridge design by a computer. The weight computation, cost estimate, and many other items can be automatically tabulated.

In various engineering fields, many application packages (subsystems) are often assembled together. A typical user writes in a problem-oriented language to call one of the subsystems. Typical examples include ICES (Integrated Civil Engineering System), TIES (Total Integrated Engineering System), and ECAP (Electrical Circuit Analysis Program). These large programming systems are designed to widen the base of computer usage in the engineering profession.

The most common subsystems used in ICES are :

1. COGO (Coordinate Geometry), which solves geometric problems in civil engineering.
2. STRUDL (STRUctural Design Language), which performs structural analysis for two- or three-dimensional frames.
3. TABLE-I, which may be used to create or edit the data sets of tabular information. The data may then be used in connection with any other subsystem.

Continuous advances in software and hardware have made possible many new innovations in computer-aided design in engineering and also have permitted a more direct partnership between computers and their human users.

Time sharing, which brings the engineer closer to the computer, is often used in practice. At his own computer terminal, each engineer executes his program in an *interactive* manner.

Another interesting development is in the area of engineering graphics. By using a **lightpen** on a display scope, as shown in Fig. 2, one can draw two projections of a given object, ask the computer to straighten out lines, or rectify angles, or replace one part of the object with a new one. A sequence of perspective views can be produced by the computer. Similarly, modifications of a surface can be made on a display scope.

Several time-sharing systems have recently been supplemented by virtual memory facilities. Virtual

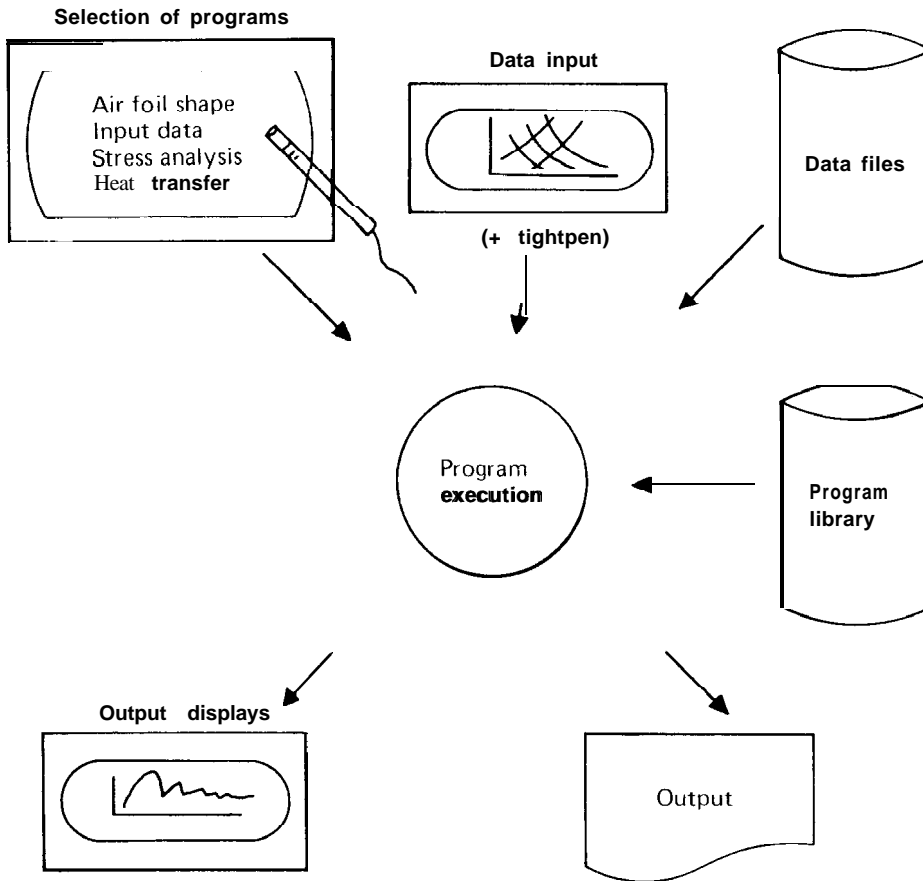


Fig. 2. Engineering design using a virtual machine time-sharing system.

memory is a technique that makes a computer appear to have considerably more main storage capacity than it actually has. This makes possible the concurrent processing of programs that in total size would exceed main storage capacity.

The development of a virtual machine time-sharing system with display consoles has been useful in computer-aided design. It allows an engineer to make the computer an integral part of an engineering design sequence through the interactive usage of display consoles. A typical example is the design of jet engine turbine blades. The following steps might be involved:

1. Sitting at a display console, the design engineer calls for a program for airfoil shape. Initial geometric coordinates may be read in through a card reader. The points or lines of the resulting picture on a display scope may then be added, changed, or deleted.

2. The engineer calls for a program such as one to perform static or dynamic stress analysis. The result is graphically shown on the display console. At this point, he may wish to go back to step 1 or go to step 3.

3. The design engineer calls for another program, such as a heat transfer program, and so on.

These steps can be followed in Fig. 2.

Conclusion. We conclude our discussion by pointing out some development problems associated with engineering applications.

At the present time, application packages in some engineering disciplines have been developed with relatively little consideration given to language, hardware and software environment, documentation, and maintenance. The growth of these packages is extensive, but it is unmonitored and uncontrolled. As a result, problems of availability and

usability of some packages do exist. Concentrated efforts, however, to distribute packages in various fields have been made. They include the following groups in the United States:

- APEC (Automated Procedures for Engineering Consultants)
- CEPA (Civil Engineering Programming Applications)
- COSMIC (Computer Software Management and Information Center)
- STORE (STRUCTURES ORiented Exchange).

Computer-oriented standards and procedures are also being developed in the United Kingdom for engineering software; these are known as GENESYS (GENERAL Engineering SYSTEM).

The usability of engineering software packages is measured by the three related factors: (1) portability; (2) adaptability; and (3) maintenance.

The quality of implementation involves: (1) documentation; (2) modular structure; (3) source language; (4) self-checking mechanisms in the programs, and (5) test data provided with the programs.

How to effectively increase the usability of application packages represents a constant challenge to the engineering profession.

REFERENCES

- 1967. Kuo, S. S. *Computer Analysis Of Orthotropic Steel Plate Superstructures for Highway Bridges*, (4 vol., PB-173 355 through PB-173 358). Springfield, Va. : U.S. Department of Commerce, National Technical Information Service.
- 1972. Furman, T. T. (Ed.). *The Use Of Computers in Engineering Design*. New York: Van Nostrand-Reinhold.
- 1972. Kuo, S. S. *Computer Applications of Numerical Methods*. Reading, Mass. : Addison-Wesley.

S. S. KUO

ENIAC

For articles on related subjects see **DIGITAL COMPUTERS**: Early; **ECKERT, J. PRESPEER**; and **MAUCHLY, JOHN W.**

The ENIAC (Electronic Numerical Integrator and Computer) was developed at the Moore School of the University of Pennsylvania in Philadelphia

between 1943 and 1946. It was the first electronic automatic computer, and it was certainly a landmark leading to the development of many automatic computer designs. The logical design of the system was based on the ideas of John Mauchly, and credit for the engineering goes to J. Presper Eckert, Jr.

The ENIAC was literally a giant. It contained more than 18,000 vacuum tubes, weighed 30 tons, and occupied a room 30 by 50 feet.

The computer consisted of 20 electronic accumulators, multiplier control, divider and square root control, input, output, two function tables, and a master program control. Each accumulator could store, add, and subtract lo-decimal digit numbers. Two accumulators could be interconnected to perform 20 digit operations. Addition and subtraction took 200 μ s. Multiplication involved six accumulators and took 2,600 μ s.

Decimal digits were stored in ten-stage ring counters, and signed decimal numbers were transmitted in parallel over 11 lines. Each digit was represented during transmission by a train of 0 to 9 pulses. Clock rates were 100 kHz and pulse widths about 2 μ s. All logic was accomplished with direct-coupled vacuum tube circuitry.

As initially designed, programming was by patch panel interconnection, with a wire being required for each event at each unit. Data paths were programmable, using 11 wire cables. The data paths were like a party-line telephone-many units could listen, but only one could transmit. Various units could operate in parallel, being initiated from the same program signal and perhaps using distinct data paths. Interlocks were provided so that independent actions of indeterminate length (e.g., card reading) could complete before follow-on actions were initiated. Signs of results could change the flow of control.

The ENIAC was converted later to a card-programmed computer. In this scheme, certain standard operations were set up in the patch panel wiring, and sequences of these macro operations were initiated from the card reader.

The ENIAC was designed to integrate ballistic equations, and a significant accomplishment at its dedication in February 1946 was the computation of the trajectory of a 16-inch naval shell in less than real time. It was formally accepted a few months after its dedication by the U.S. Army Ordnance Corps, but was still operated at the Moore School until late 1946, when it was dismantled and shipped to Aberdeen Proving Ground in Maryland. It became operational again in 1947, and was operated until Oct. 2, 1955 (Weik, 1961, p. 575).

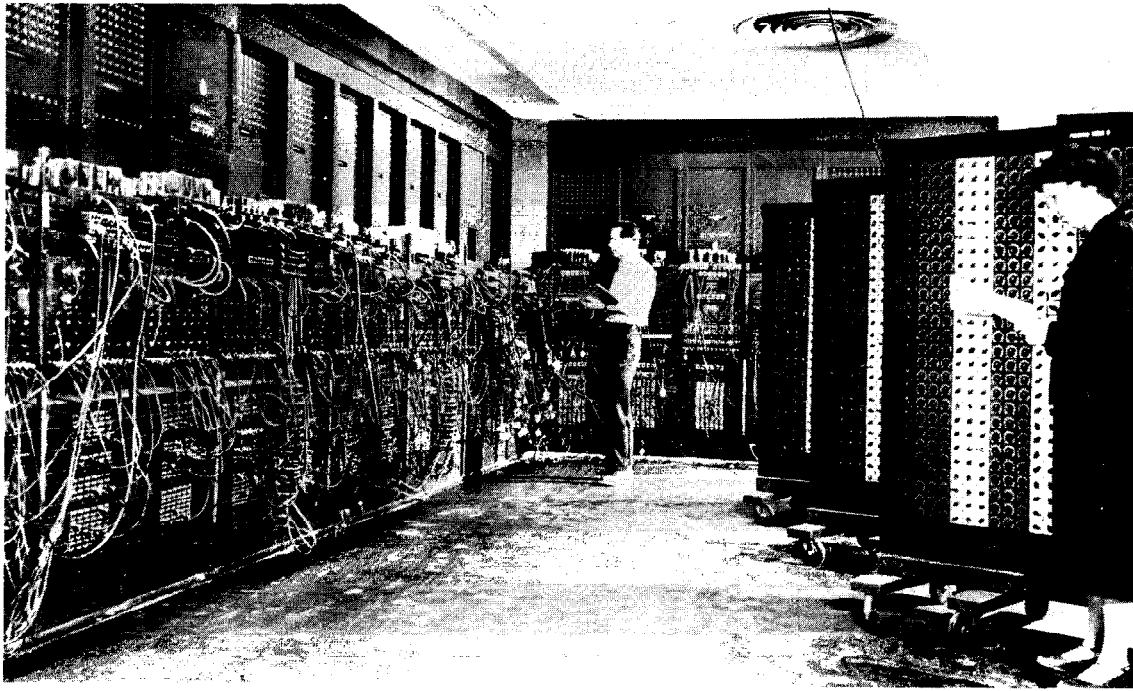


Fig. 1. ENIAC. (Courtesy of Smithsonian Institution.)

The first significant computation on the ENIAC involved atomic energy. Since World War II had ended, there was no longer urgent need for the firing tables that had motivated its design and the support of the Army Ordnance Corps. Among the problems first computed on it, in addition to those involving atomic energy, were random number studies, round-off errors, cosmic ray studies, thermal ignition, wind tunnel design, and weather prediction. It was the major instrument for the computation of all ballistic tables for the U. S. Army and Air Force (Weik, 1961).

Aberdeen Proving Ground reported that during 1952 the "total machine time" for the ENIAC was 7,247 hr., divided as follows: production, 3,491 hr; problem setup and code-checking, 1,061 hr; idle, 195.3 hr; scheduled engineering, 65.1 hr; and unscheduled "engineering," 1,847.8 hr. The major portion of the scheduled engineering was preventive servicing, the remainder being for improvements and additions; 90% of the unscheduled engineering was devoted to locating and replacing defective tubes. During 1952 approximately 19,000 tubes were replaced (more than 100% of the tube complement).

The ENIAC proved that, with careful engineering, it was possible to build extremely complex logical devices that would perform at electronic

speed, without error, for significant periods of time. This was the landmark leading to the development of many automatic computer designs, and paving the way for the "computer revolution." As modestly noted by the Ordnance Corps in *Army Ordnance* (1946), the ENIAC "established the fact that the basic principles of electronic engineering are sound." It was indeed "inevitable that future computing machines of this type would be improved through the knowledge and experience gained on this first one."

Portions of the ENIAC are now in the Smithsonian Institution at Washington, D.C. Other ENIAC materials are in the custody of the Historical Services Division of the Department of the Army in Washington.

REFERENCES

- 1946. U.S. Army Ordnance Corps. "Mathematics by Robot," *Army Ordnance*, Vol. XXX, No. 156 (May-June), pp. 329-331.
- 1950. Stiffler, W. W., Jr. (Ed.). *High Speed Computing Devices*. New York: McGraw-Hill.
- 1961. Weik, Martin H. "The ENIAC Story," *Army Ordnance*, Vol. XLV, No. 244 (January-February), pp. 571-575.

H. D. HUSKEY

EQUATIONS

EQUATIONS. See **PARTIAL DIFFERENTIAL EQUATIONS, NUMERICAL SOLUTION OF**; and **NUMERICAL ANALYSIS**.

ERROR ANALYSIS

For articles on related subjects see **ARITHMETIC, COMPUTER**; **ERRORS**; **ERRORS, ABSOLUTE AND RELATIVE**; **INTERVAL ARITHMETIC**; **MATRIX COMPUTATIONS**; **NUMERICAL ANALYSIS**; **ROUND-OFF ERROR**; and **SIGNIFICANCE ARITHMETIC**.

In general the basic arithmetic operations on digital computers are not exact but are subject to rounding or truncation errors. This article is concerned with the cumulative effect of these errors. It will be assumed that the reader has read the article on Matrix Computations since the results will be illustrated by examples from that area.

Definitions. There are two main methods of error analysis, known as "forward analysis" and "backward analysis," respectively. They may be illustrated by considering the solution of an $n \times n$ system of linear equations by Gaussian elimination. In this algorithm, the original system is reduced successively to equivalent systems $A^{(r)}x = b^{(r)}$, $r = 1, 2, \dots, n-1$. In the final system the matrix of coefficients, $A^{(n-1)}$ is upper-triangular, and the solution is found by a back substitution.

In a forward analysis, one adopts the following strategy: Because of rounding errors the computed derived system $\bar{A}^{(r)}x = \bar{b}^{(r)}$ differs from that which would be obtained by exact arithmetic. It seems reasonable to assume that if the algorithm is stable, $\bar{A}^{(r)} - A^{(r)}$ and $\bar{b}^{(r)} - b^{(r)}$ will be small, and with sufficient ingenuity bounds would be found for these "errors." This is perhaps the most natural approach.

Alternatively, one could adopt the following strategy: If the algorithm is stable, presumably the computed solution \bar{x} is the exact solution of some system $(A + E)\bar{x} = b + e$, where E and e are relatively small. Of course there will be an infinite number of sets, of which \bar{x} is the exact solution. A successful error analysis will obtain satisfactory bounds for the elements of E and e . Such an approach is known as "backward" error analysis, since it seeks to replace all errors made in the course of the solution by an *equivalent* perturbation of the

original problem. It has one immediate advantage. It puts the errors made during the computation on the same footing as those arising from the data. Hence, when the initial data is itself inexact, no additional problem is posed.

Early Error Analysis of Elimination Processes. In the 1940s the imminent arrival of electronic computers stimulated an interest in error analysis, and one of the first algorithms to be studied was Gaussian elimination. Early analyses were all of the forward type, and typical of the results obtained was that of Hotelling, who showed that errors in solving an $n \times n$ system might build up by a factor 4^{n-1} . The relevance of this result was widely accepted at the time. Writing in 1946, Bargmann, Montgomery, and von Neumann said of Gaussian elimination: "An error at any stage affects all succeeding results and may become greatly magnified; this explains why instability should be expected." The mood of pessimism was very infectious, and the tendency to become enmeshed in the formal complexity of the algebra of the analysis seems to have precluded a sound assessment of the nature of the problem. Before giving any error analysis we discuss fundamental limitations on the attainable accuracy.

Norms and Floating-Point Arithmetic. We will need some way of assessing the "size" of a vector or a matrix. Such a measure is provided by vector and matrix *norms*. A norm of a vector x , denoted by $\|x\|$, is a nonnegative quantity satisfying the relations

$$\begin{aligned}\|x\| &\geq 0 \quad \text{and} \quad \|x\| = 0 \quad \text{iff } x = 0, \\ \|\alpha x\| &= |\alpha| \|x\|, \\ \|x + y\| &\leq \|x\| + \|y\|.\end{aligned}$$

We will use only two norms, denoted by $\|x\|_2$ and $\|x\|_\infty$ and defined by

$$\|x\|_2 = (\sum |x_i|^2)^{1/2}, \quad \|x\|_\infty = \max |x_i|.$$

Similarly, a norm of a matrix A , denoted by $\|A\|$, is a nonnegative quantity satisfying the relations

$$\begin{aligned}\|A\| &\geq 0 \quad \text{and} \quad \|A\| = 0 \quad \text{iff } A = 0, \\ \|\alpha A\| &= |\alpha| \|A\|, \\ \|A + B\| &\leq \|A\| + \|B\|, \\ \|AB\| &\leq \|A\| \|B\|.\end{aligned}$$

We will use only two norms, denoted by $\|A\|_2$ and $\|A\|_\infty$ and defined by

$$\|A\|_2 = (\max \text{eigenvalue of } A^T A)^{1/2},$$

$$\|A\|_\infty = \max_i (\sum_j |a_{ij}|).$$

It may be verified that

$$\|Ax\|_2 \leq \|A\|_2 \|x\|_2$$

$$\|Ax\|_\infty \leq \|A\|_\infty \|x\|_\infty.$$

Most of the early error analyses were for fixed-point computation, but since virtually all scientific computation is now done in floating point, we restrict discussion to this case. We use the notation $\text{fl}(x \times y)$ to denote the product of two standard floating-point (fl) numbers as given by the computer under examination, with an analogous notation for the other arithmetic operations. We have the following results for each of the basic operations, using a mantissa of t digits in the base β :

$$\begin{aligned} \text{fl}(x \times y) &= xy(1 + E), & |E| &\leq m\beta^{-t}, \\ \text{fl}(x \div y) &= (x/y)(1 + E), & |E| &\leq d\beta^{-t}, \\ \text{fl}(x \pm y) &= x(1 + \epsilon_1) \pm y(1 + \epsilon_2), \\ & & |\epsilon_1|, |\epsilon_2| &\leq s\beta^{-t}, \end{aligned}$$

where m , d , and s are constants on the order of unity, depending on the details of the rounding or chopping procedure. Described in the language of backward errors analysis, we might say, for example, that the *computed* sum of two numbers x and y is the exact sum of two numbers $x(1 + \epsilon_1)$ and $y(1 + \epsilon_2)$, each having a low relative error. On well-designed computers,

$$\text{fl}(x \pm y) = (x \pm y)(1 + E), \quad |E| \leq s\beta^{-t}.$$

For convenience from now on we assume that all ϵ in the above satisfy the bound $|\epsilon| \leq k \cdot \beta^{-t}$, where k is of the order of unity.

By repeated application we have, with an obvious notation,

$$\begin{aligned} \text{fl}(a_1 + a_2 + \dots + a_n) \\ = a_1(1 + E_1) + a_2(1 + E_2) + \dots + a_n(1 + E_n), \\ (1 - k\beta^{-t})^{n-1} \leq 1 + E_i \leq (1 + k\beta^{-t})^{n-1}, \\ (1 - k\beta^{-t})^{n+1-t} \leq 1 + E_r \leq (1 + k\beta^{-t})^{n+1-t} \\ r = 2, 3, \dots, n. \end{aligned}$$

The bounds on the errors are reasonably realistic and examples can be constructed in which they are almost attained. Naturally, when n is large, the statistical distribution can be expected, in general, to result in some cancellation of errors and, thus, in actual errors substantially less than the bounds.

One of the most important elements in elimination methods is the computation of expressions of the form

$$p = \text{fl}(a - x_1 \times y_1 - \dots - x_n \times y_n).$$

The computed p and the error bounds are dependent on the order in which operations are performed. If the operations are performed in the order written above, we obtain

$$p = a(1 + E) - x_1 y_1(1 + F_1) - \dots - x_n y_n(1 + F_n),$$

where

$$\begin{aligned} (1 - k\beta^{-t})^n &\leq 1 + E \leq (1 + k\beta^{-t})^n, \\ (1 - k\beta^{-t})^{n+2-i} &\leq 1 + F_i \leq (1 + k\beta^{-t})^{n+2-i}. \end{aligned}$$

If one computes

$$P = \text{fl}(-x_1 \times y_1 - x_2 \times y_2 - \dots - x_n \times y_n + a),$$

then

$$\begin{aligned} P &= -x_1 y_1(1 + E_1) - \dots - x_n y_n(1 + E_n) + a(1 + F), \\ (1 - k\beta^{-t})^{n+3-t} &\leq 1 + E_i \leq (1 + k\beta^{-t})^{n+3-t}, \\ |F| &\leq k\beta^{-t}. \end{aligned}$$

In describing the last result in terms of backward error analysis, we might say, for example, that it is exact for data $x_i(1 + E_i)$, y_i , and $a(1 + F)$, putting all the perturbations in the x_i and a . Alternatively, we could say it is exact for data x_i , $y_i(1 + E_i)$, and $a(1 + F)$.

Note that although the errors made can be equated with the effect of small relative perturbations in the data, the relative error in the computed p may be arbitrarily high, depending on the degree of cancellation that takes place. Indeed, if the true p is zero, one may have an infinite relative error. One would not think of attributing this to some malignant instability in this simple arithmetic process; it is the natural loss to be expected.

Inherent Sensitivity of the Solution of a Linear System. For any computational problem the inherent sensitivity of the solution to changes in the data is of fundamental importance, yet oddly enough the early analyses of Gaussian elimination paid little attention to it. We consider in a very elementary way the effect of perturbations δA in the

ERROR ANALYSIS

matrix A . We have

$$\bar{x} = (A + \delta A)^{-1}b = (A^{-1} - A^{-1}\delta A A^{-1} + \dots)b \\ = x - A^{-1}\delta A x + (A^{-1}\delta A)^2 x - \dots,$$

giving

$$\|\bar{x} - x\|/\|x\| \leq \|A^{-1}\delta A\|/(1 - \|A^{-1}\delta A\|),$$

provided $\|A^{-1}\delta A\| < 1$. The relative error in \bar{x} will not be low unless $\|A^{-1}\delta A\|$ is small. Writing,

$$\|\delta A\| = \eta \|A\|,$$

we see that

$$\|\bar{x} - x\|/\|x\| \\ \leq \eta \|A\| \|A^{-1}\|/(1 - \eta \|A\| \|A^{-1}\|).$$

The inherent sensitivity is therefore dependent on $\|A\| \|A^{-1}\|$, and this is usually known as the "condition number" of A (for the given norm) with respect to inversion or to the solution of linear systems.

We might now ask ourselves what sort of limitation we should expect on the accuracy of Gaussian elimination even if it had no menacing instability. The solution of $Ax = b$ requires $n^3/3$ multiplications and additions, an average of $1/3n$ per element. From the elementary discussion given so far, we might risk the following prophecy: Even if Gaussian elimination is a stable process, then we can scarcely expect to obtain a bound for the resulting error, which is less than that resulting from a perturbation δA in A satisfying, say,

$$\|\delta A\| \leq \frac{1}{3}kn\beta^{-1}\|A\|.$$

In fact, this bound for the effect is usually reasonably realistic, provided pivoting is used. Indeed, the advantages conferred by the statistical distribution of rounding errors is such that the error is usually less than the maximum error that could be caused by such a perturbation.

Backward Error Analysis of Gaussian Elimination. Gaussian elimination provides a very good illustration of the power and simplicity of backward error analysis. The elimination process may be described as the production of a unit lower triangular matrix L and an upper triangular matrix U such that $LU = A$. The solution of the system $Ax = b$ is then carried out in the two steps:

$$Ly = b, \quad Ux = y.$$

In the backward error analysis one shows that the computed L and U satisfy the relation $LU = A + E$ and obtains bounds for the elements of E . One then shows that the computed solution x of the triangular systems satisfies the equations

$$(L + \delta L)y = b, \quad (U + \delta U)x = y$$

and obtains bounds for the elements of δL and δU . The computed x therefore solves exactly the system

$$(L + \delta L)(U + \delta U)x = b$$

or

$$(A + E + \delta LU + L\delta U + \delta L\delta U)x = b.$$

Hence, it is the exact solution of $(A + F)x = b$, where

$$\|F\| = \|E + \delta LU + L\delta U + \delta L\delta U\| \\ < \|E\| + \|L\| \|\delta U\| \\ + \|U\| \|\delta L\| + \|\delta L\| \|\delta U\|,$$

and from the bounds for E , δL , and δU , one obtains a bound for F .

The simplicity of the technique may be illustrated by presenting the analysis of the solution of the system $Ly = b$. We first make the following observations:

1. The relevant system to be analyzed is that with the computed matrix L , *not* the L that would have resulted from exact computation.
2. Since during the course of the analysis we do not attempt a direct comparison between computed and exact values, there is no need to denote computed quantities by bars. It is to be understood that all symbols refer to computed quantities.
3. It is only at the final stage when we have expressed the computed solution as the exact solution of $(A + F)x = b$ and have obtained a bound for $\|F\|$ that we attempt to compare the computed x with the true x , and at this stage we can use the result of the previous section.

At a typical stage in the triangular solution, y_1, y_2, \dots, y_{r-1} have been computed and y_r is determined from the relation

$$y_r = \text{fl}(-l_{r,1}y_1 - l_{r,2}y_2 - \dots - l_{r,r-1}y_{r-1} + b_r),$$

using, of course, the computed values of the y_i . Hence

$$y_r = -l_r y_1(1 + E_{r,1}) - l_r y_2(1 + E_{r,2}) - \dots - l_{r,r-1} y_{r-1}(1 + E_{r,r-1}) + b_r(1 + F_r),$$

where the factors $1 + E_{r,i}$ and $1 + F_r$ are of the type discussed in connection with the computation of p above. Hence, the computed y_i satisfy exactly the relation

$$l_r y_1(1 + G_{r,1}) + l_r y_2(1 + G_{r,2}) + \dots + l_{r,r-1} y_{r-1}(1 + G_{r,r-1}) + y_r(1 + G_{r,r}) = b_r,$$

where

$$(1 + G_{r,i}) = (1 + E_{r,i})/(1 + F_r), \quad i = 1, \dots, r-1, \\ 1 + G_{r,r} = 1/(1 + F_r).$$

Notice that by dividing through by $1 + F_r$, we are able to restrict ourselves to perturbations in L . The computed y therefore satisfies exactly the relation $(L + \delta L)y = b$, where $\delta L_{ij} = L_{ij} G_{ij}$.

We certainly have

$$(1 - k\beta^{-t})^n \leq (1 + G_{ij}) \leq (1 + k\beta^{-t})^n,$$

most of the factors, of course, satisfying much better bounds. Bounds of the above type are cumbersome to use, and we observe that if $kn\beta^{-t} < 0.1$, as will usually be the case, then, using the binomial theorem,

$$(1 + k\beta^{-t})^n \leq 1 + (1.06)kn\beta^{-t}, \\ (1 - k\beta^{-t})^n \geq 1 - (1.06)kn\beta^{-t}.$$

Hence, we have

$$|\delta L_{ij}| \leq (1.06)kn\beta^{-t}|L_{ij}|,$$

giving, for example,

$$\|\delta L\|_\infty \leq (1.06)kn\beta^{-t}\|L\|_\infty.$$

The analysis is almost trivial, though earlier error analyses of the solution of triangular systems were extremely complicated.

If the computation of y_r had been expressed in the form

$$y_r = \text{fl}(b_r - l_r y_1 - \dots - l_{r,r-1} y_{r-1}),$$

then we could still obtain a relation of the form $(L + \delta L)y = b$, but in this case the bounds on the elements of δL would be appreciably larger.

On many computers it is possible to accumulate either of the expressions for y_r in double precision, rounding to single precision only on completion. If this is done, then we again obtain a relation of the form

$$l_r y_1(1 + G_{r,1}) + l_r y_2(1 + G_{r,2}) + \dots + l_{r,r-1} y_{r-1}(1 + G_{r,r-1}) + y_r(1 + G_{r,r}) = b_r,$$

but now the quantities $|G_{r,i}|$ ($i < r$) have bounds of order β^{-2t} and can therefore virtually be neglected, while $|G_{r,r}|$ has the bound $k\beta^{-t}$. We therefore have a result that might well be described as best possible, having regard to the precision of computation. Indeed, the residual vector $b - Ly$ corresponding to the computed y will almost certainly be smaller than that corresponding to the correctly rounded solution!

The analysis of the solution of $Ux = y$ is almost identical to that of $Ly = b$, while the analysis of the factorization process is only marginally more complicated. If the L and U are produced as in classical Gaussian elimination, then one can show that $LU = A + E$, where, denoting the maximum modulus of any element arising during the decomposition by g , we certainly have

$$|e_{ij}| \leq (3.02)igk\beta^{-t} \quad (i \leq j), \\ |e_{ij}| \leq (3.02)jgk\beta^{-t} \quad (i > j).$$

If the factors L and U are determined directly, using the relations

$$l_{ij}u_{jj} = a_{ij} - l_{i1}u_{1j} - \dots - l_{i,j-1}u_{j-1,j} \quad j = 1, \dots, i-1$$

and

$$u_{ij} = a_{ij} - l_{i1}u_{1j} - \dots - l_{i,i-1}u_{i-1,j} \quad j = i, \dots, n,$$

and the expressions on the right are accumulated in double precision, an even more satisfactory bound may be determined for E . Indeed, ignoring quantities of the order of magnitude of β^{-2t} , we certainly have $|e_{ij}| \leq gk\beta^{-t}$, where g is now the element of maximum modulus in the computed U . Again, we have what may be regarded as a "best possible" result.

The reader may be surprised that no reference has been made to pivoting or to the size of the l_{pq} . The importance of pivoting is concealed. If any of the multipliers is large, g will usually be much larger than $\max |a_{ij}|$. When pivoting is used $|l_{pq}| \leq 1$, and there will not *usually* be much growth in the size of the elements of the reduced matrices or of U relative to the initial set of a . When A is positive definite or diagonally dominant, *no* growth can take place, and we have a guaranteed a priori bound for $\|E\|$ in terms of A .

In 1947 von Neumann and Goldstine considered the special case of the inversion of a positive definite matrix with pivoting, and obtained a result for fixed point computation which is only marginally weaker than can be obtained by arguments of the above type, though the analysis was far more complicated. Their analysis is often described as a "forward error analysis," but it is in fact of the backward type, although at no stage are results expressed in a form such as to emphasize this. The final results of an analysis of the above type for the solution of a positive definite system is to guarantee that it is the exact solution of $(A + E)x = b$ and to give a bound for E of the type

$$\|E\| \leq f(n)k\beta^{-1}\|A\|,$$

where $f(n)$ is a modest function of n , depending a little on the details of the arithmetic. When backward error analysis is applied to matrix inversion, one cannot show that X is the exact solution of $(A + E)X = I$, with a similar bound for E , because it is not true. However, the r th column, x_r , of X is the exact solution of some $(A + E_r)x_r = e_r$, where e_r is the r th column of I ; the E_r are all different, but have the same satisfactory uniform bound. This result is implicit in that of von Neumann and Goldstine, but it is well concealed!

Orthogonal Transformations. Experience with error analyses of matrix processes gradually exposed the fact that control of *growth* in derived matrices is the key to stability. If orthogonal transformations Q are used, then since $\|QA\|_2 = \|A\|_2$, $\|Q\|_2 = 1$ —no general growth *can* take place. Although the algebra is a little complicated, a fairly general analysis can be given of whole classes of algorithms based on orthogonal transformations, both for the solution of equations and the eigenvalue problem. One can show, for example, that for a sequence of r orthogonal similarity transformations, the final computed transform $A^{(r)}$ satisfies *exactly* a relation of the form

$$A^{(r)} = Q^T(A + E)Q,$$

where Q is *exactly* orthogonal and

$$\|E\| \leq rf(n)\|A\|k\beta^{-1},$$

where $f(n)$ is some quite innocuous function of n . Hence, the eigenvalues of $A^{(r)}$ are exactly those of $A + E$ and we are back with perturbation theory.

A Posteriori Error Bounds. The bounds discussed so far are of the a priori type. The main function of such an analysis is to show whether or not an algorithm is stable and, if not, to pinpoint the reasons for its instability.

When a solution has been determined, one can usually obtain much sharper backward error bounds. For example, from a computed eigenvalue λ and an eigenvector u , such that $\|u\|_1 = 1$, one can compute the residual defined by $r = Au - \lambda u$. This may be written in the form $(A - ru^H)u = Au$, showing that λ and u are exact for the matrix $A - ru^H$. When A is Hermitian, this implies that A has an eigenvalue in the interval $\lambda - \|r\|_2, \lambda + \|r\|_2$. Similarly, when solving linear equations one can compute $r = b - Ax$. If r is computed accurately, it can then be used to obtain an improved solution by solving $A\delta = r$. This process is called "iterative refinement."

Iterative Methods. It was at one time thought that iterative methods for solving linear equations or the eigenvalue problem would give far greater accuracy than direct methods, since one works with the initial A throughout. In fact this advantage is largely illusory. In Jacobi's method for linear equations, one derives an improved $x_i^{(r+1)}$ from the relation

$$a_{ii}x_i^{(r+1)} = b_i - \sum_{j \neq i} a_{ij}x_j^{(r)},$$

but the right-hand side cannot be computed exactly. From the above analysis it is clear that one is really working with a matrix with elements $a_{ij}(1 + e_{ij})$, where the e_{ij} are different in each iteration. When iterative methods are used in practice, iteration is usually terminated before attaining the accuracy given immediately by a direct method, even *without iterative refinement*. Since, as we mentioned earlier, the results obtained with good direct methods are almost "best possible," this is to be expected.

Interval Arithmetic and Significant Digit Arithmetic. Attempts have been made to obtain

error bounds for computed quantities on the computer itself. In *interval* arithmetic, an ordered pair $[a_l, a_u]$ of floating-point numbers is stored at each stage in the computation, and it is guaranteed that the true number a lies in the interval $a_l \leq a \leq a_u$. Used in a direct manner, the results achieved are very pessimistic; in fact, the computer merely performs numerically the analog of what was done algebraically in the early forward error analysis of the **Hotelling** type. The intervals become very large. The apparently reasonable assumption that in stable algorithms the computed quantities will be close to those arising in exact computation is frequently quite false. This is particularly true of algorithms for the eigenvalue problem.

In *significant digit* arithmetic, one does not work with normalized floating-point numbers, on the grounds that when cancellation takes place, the zeros introduced are nonsignificant. The possibilities of significant digit arithmetic have been well exploited by Metropolis and Ashenhurst.

The realization that neither interval arithmetic nor significant digit arithmetic provides an automatic answer to error analysis led to an overreaction against them. The provision of the relevant hardware facilities should make them economic, and when combined with a more general appreciation of theoretical error analysis, they have an important role to play.

REFERENCES

- 1963. Wilkinson, J. H. *Rounding Errors in Algebraic Processes*. London: Her Majesty's Stationery Office.
- 1965. Wilkinson, J. H. *The Algebraic Eigenvalue Problem*. Oxford: Clarendon Press.
- 1966. Moore, R. E. *Interval Analysis*. Englewood Cliffs, N.J.: Prentice-Hall.
- 1967. Forsythe, G. E., and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, N.J.: Prentice-Hall.

J. H. WILKINSON

ERROR-CORRECTING CODE

For articles on related subjects see **CODES**; and **ERRORS**.

For article on related term see **PARITY**.

Error-detecting and error-correcting codes arose from the well-known phenomenon that if anything can go wrong, it will. Rather than try to do everything perfectly the first time, error-detecting and error-correcting methods use some form of redundancy to handle the inevitable errors.

Error detection has a long history. For example, suppose we have a block of n binary digits and add an $(n + 1)$ st digit, chosen so that the whole message has an even (or odd) number of 1s in it. This is called an even (odd) parity check. At the receiving end, the complete block is checked. If there are not the proper number of 1s in the message, then there must be an odd number of errors in the message. If the block is chosen to be reasonably short (with respect to the probability of an isolated error), so that we may ignore factors of $1 - p$, and if we assume that errors are independent? then to a close approximation there is a probability $(n + 1)p$ of a single error, and a probability $\{n(n + 1)/2\}p^2$ of two errors,

Upon the detection of an error, the message can be retransmitted, and generally this will produce an error-free message. In some circumstances, especially where it is suspected that the source is slightly defective (say, a magnetic recording), several retrials may be used before giving up. The retrial system is not entirely satisfactory because it takes extra time when errors occur and also requires two-way signaling to call for message repetition. However, if the error is in the original recorded form of the message before encoding, then nothing can be done about the error.

To overcome these difficulties, error-correcting codes are often used. They are based on the use of a high level of redundancy, i.e., repeated parity checks.

There are various ways of explaining how an error-correcting code works. In the algebraic approach, a parity check is assigned to those positions in the code that have a 1 in the rightmost position of their binary representation, a second parity check for those positions that have a 1 in their second to right position, etc. Thus, when a single error does occur, exactly those parity checks will fail for which the binary expansion of the position of the error has 1s. Thus, the pattern of the parity-check failures points directly to the position of the error; in a binary system of signaling, it is easy to change that bit to its opposite value and thus correct the error, with 000 meaning "no error."

As an example, consider the binary encoding of the decimal digits into an error-correcting code. In Table 1, positions 1, 2, and 4 are used for the check positions, leaving positions 3, 5, 6, 7 for the message

ERRORS

(where we find the binary coding of the corresponding decimal digit).

Table 1

Decimal	Position						
	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1

The check positions are calculated by even parity checks as follows:

Parity check column 1

Columns 1, 3, 5, 7 (columns with a 1 in the rightmost position of their binary representation).

Parity check column 2

Columns 2, 3, 6, 7 (1 in second rightmost position).

Parity check column 4

Columns 4, 5, 6, 7 (1 in leftmost position).

Let any line be copied and a single error inserted as a simulation of an error in message transmission. When the three parity checks are applied, we will find that if we write a 0 for successful parity check and a 1 for a failure (writing from right to left), the three digits we get will be exactly the position of the inserted error.

A second way of looking at the codes is a geometric approach. If an error is to be detected, then the distance between two messages (which we define to be the number of positions for which they differ) must be at least two for every pair of messages. Otherwise, there would be a message that a single error would carry over into another acceptable message, and that error could not be detected. For error correction, the minimum distance must be at least three (as in Table 1); for double error detection, the minimum distance must be at least four; etc.

The encoding process can thus be extended further in protecting against errors. As an example of double-error detection, consider the code in Table 1 with an additional bit added to each message, so

chosen that the entire message will have an even number of 1's. If there were a single error, the original set of checks would indicate the position, but the last check would fail. If there were a pair of errors, the last check would not fail, but some of the original checks would, indicating a double error. The minimum-distance argument can be applied to show that the additional check made each minimal distance one greater, namely, now four.

The preceding examples are the simplest cases. The theory has been highly developed and now makes use of much of abstract algebra, including Galois theory.

REFERENCES

1961. Peterson, W. W. *Error Correcting Codes*. Cambridge, Mass.: The M.I.T. Press.
1968. Berlekamp, E., Jr. *Algebraic Coding Theory*. New York: McGraw-Hill.

R. W. HAMMING

ERRORS

For articles on related subjects see DEBUGGING; DIAGNOSTICS; ERRORS, ABSOLUTE AND RELATIVE; ERROR ANALYSIS; ROUND-OFF ERROR; STRUCTURED PROGRAMMING; and SYNTAX, SEMANTICS, AND PRAGMATICS.

For articles on related terms see ALGORITHM; ARITHMETIC-LOGIC UNIT; CENTRAL PROCESSING UNIT; COMPUTER SYSTEMS; GENERATIONS, COMPUTER; MICROPROGRAMMING; SOFTWARE; and VIRTUAL MEMORY.

The indignant customer who receives an incorrect bill from a department store probably does not care what the source of the error was or even that, almost certainly, the fault was not the computer's but its programmer's. Neither is the astronaut descending toward the surface of the moon very concerned about the precise source of the error that caused his on-board computer to fail. But an understanding of the sources of errors in computers is important to anyone who wishes to use or even to comprehend digital computers.

TAXONOMY OF COMPUTER ERRORS. When a computer produces an incorrect result, the error may

come from one or more of a number of sources. These sources can be fairly readily grouped under three headings:

Hardware errors, which result from a malfunction of some physical component of the computer.

Software errors, which result from a coding error in *some* program, but not necessarily in the program that seemed to produce the wrong results (see below).

Algorithm errors, which result when the algorithm or method used to solve a problem does not produce correct results, perhaps only under certain conditions and/or for certain input data.

Before proceeding to discuss these three types of errors in some detail, we should stress that, whereas in the early days of computing it was usually rather easy to determine which of the three categories above was the source of an error, it is sometimes very difficult indeed to do this today. To give one example, the increasing use of microprogramming in contemporary computer systems makes it possible for hardware errors to manifest themselves in ways that look like software errors, and vice versa. The difficulty of determining the source of a computer error has heightened the need for good diagnostic techniques, a subject we consider in the last section of this article.

Hardware Errors. Considering the staggering complexity of modern computer systems, it is amazing that they work at all. The fact that they are designed to, and often do, operate for hundreds or thousands of hours without failure is even more startling. Modern computers contain literally millions of circuit elements, the failure of any one of which might cause failure of the entire system. This high level of reliability is a tribute to the careful work of circuit designers and the meticulous attention to detail and to testing on the part of the manufacturers. Still, computers are not perfect and the hardware occasionally does fail. The source of a failure may be difficult to determine, since the number of possible faulty components is so large.

A frequent source of errors is in the electro-mechanical peripheral devices that provide input or output for the central processing unit. The mechanical components of these peripheral devices are likely to wear out as a result of the stresses of frequent use. The staccato motion of movable disk arms, the rapid rotation of disk packs, drums, or tape drives, the stop-and-go movements in card readers and punches, all are possible sources of failures.

The recording medium associated with each of these devices is fragile and consequently a potential source of errors. The delicate magnetic coating of magnetic tapes, disks, or drums can be easily scratched, rendering the information incorrect or inaccessible. A speck of dust or dirt can mar these coatings easily, or tension can stretch a piece of magnetic tape. Punched cards may be folded, spindled, or mutilated, and paper tape reels may easily be torn. The failure of these media may not be fatal to the entire computer, but individual peripheral units may be disabled or data items may be entered incorrectly or lost. Telecommunication devices attached to a computer may also be faulty. Since the quality of the voice-grade telephone lines often used for communication with computers is low, special leased lines are sometimes used to reduce the frequency of errors.

The central processing unit, arithmetic and logical unit, and the high-speed memory are built entirely from electronic components, thereby reducing the chance of failure inherent in mechanical devices. The technology for creating the circuit elements involved in these components is extremely complex. Early computers used vacuum tubes (first generation) as the primary circuit element. These large devices were relatively slow, generated a large amount of heat, required a large amount of power, and wore out easily. The invention of the transistor (second generation) in 1947 made it possible to construct smaller, faster, and much more reliable computers. Combining several transistors and other electronic elements into a single component, called the "integrated circuit" (third generation), enabled designers to create still faster and more reliable computers.

At present, computers are built from a smaller number of large-scale integrated circuits (fourth generation), which extends the notion of combining circuit elements. These highly reliable large-scale integrated circuits contain thousands of discrete circuit elements built into a single replaceable component. These devices are carefully tested during the many stages of a sophisticated fabrication process. Still, they may fail as a result of temperature changes, humidity, shock, or electrical surges. When failure occurs, the faulty circuit component must be located and replaced. This sounds simple enough, but the problem may be hard to locate, since the failure may be intermittent, occurring only when a complex combination of conditions exist. To minimize the deterioration of circuit elements, computer center rooms are air conditioned to keep the temperature and humidity within acceptable ranges. The

ERRORS

failure of the air conditioning would lead to overheating of circuit elements and to an increased chance of failure.

Modern computers are designed to monitor their own performance and constantly test themselves to assure that each operation has been performed properly. When a fault occurs, a machine interrupt is issued, and the hardware and software attempt to identify and locate the error. Depending on the severity of the error, the control programs may shut down the entire machine, avoid use of the faulty component, or simply record the fact that an error has occurred.

Software Errors. Anyone who has written a computer program knows that debugging can be difficult and tedious. Professionals writing short programs (say, less than 100 lines of code) expect some difficulties and accept the fact that long programs, requiring many man-years of effort, may never be completely debugged. When writing programs in a higher-level language, which require the services of a compiler, utility programs, and an operating system, the number of software modules that come into play is large. Great effort is applied to debug the system software, but it is not currently possible to insure the correctness of such sophisticated programs. If an application program does not operate correctly, the most likely source of the error is in the application program itself. Only after a thorough and careful analysis of the situation can we begin to consider the possibility that the compiler, system utilities, or operating system are at fault. Locating the bug in the system software requires a deep understanding of the code and the expertise of a systems programmer.

Application program errors fall into two basic categories: syntactic and semantic. The syntactic errors include typographic errors, incorrectly spelled keywords and variable names, incorrect punctuation and improper statement formation, all of which result from violations of the programming language syntax. These errors are normally recognized by the language processor, and diagnostic messages are printed to assist the programmer in making corrections. Although some processors will attempt to fix improper syntax, programs with syntactic errors will generally not be permitted to execute.

Assuming all the syntactic errors have been fixed, the program will execute, but there is no guarantee that it will perform as the programmer intended. Semantic errors are a result of an improper understanding of the function of certain operators or mistakes in coding of an algorithm. Typical pro-

gramming mistakes include exceeding the bounds of an array; failure to initialize variables; overflow or underflow; failure to account for special cases; attempted division by zero; illegal mixing of data types; and incorrect transfers of control. Isolating and locating the error can be a long tedious process and is a skill learned mainly through much experience.

Current research is being directed at reducing the possibility of semantic errors. Improved programming language design and sophisticated compilers are one possible answer. Educating programmers to proper program design techniques such as modularity, top-down structuring, and “go-to-free” programming will hopefully simplify the debugging process. Finally, attempts are being made to prove the correctness of programs through the use of formal mathematical techniques.

Algorithm Errors. Computer programs can be viewed as models or representations of real-world situations. Unfortunately, not all aspects of the real-world situation can be accurately represented inside a computer. Decimal quantities such as 1.2 or 6783846.678492104 may have to be approximated when stored in the memory of a binary computer. Since the initial representation is not precise, subsequent operations performed on these values may produce invalid results. The difficulty in locating such faults is that the error will manifest itself only for some sets of data. Thus, the program will produce reasonable results in most cases, but may produce erroneous results erratically.

The heart of this problem is the machine representation of values. While a 60-bit word length may provide a more accurate representation than a 36-bit word or a 32-bit word, a longer word length is not a guarantee of correctness. Since we are limited to the finite length of a computer word, the representation must be rounded off to the closest approximation possible. With each addition or multiplication the result must also be rounded off to fit the representation scheme; hence the name “round-off error.”

Another flaw in the representation of the real world occurs when an infinite process must be approximated by a finite series of steps. In summing an infinite series, repeating an iterative process (e.g., the Newton-Raphson or secant methods), or approximating derivatives by differences, the result may become increasingly exact, but is never precisely correct. Since in all these cases an infinite process is cut short and represented by a finite process, this error is called “truncation error.”

One of the central concerns of numerical analysis is to estimate the maximum roundoff and truncation errors for various algorithms. This analysis can then be used to select and design the optimum strategy for a given problem.

The goal is to avoid "unstable" algorithms that operate erratically and to identify "ill-conditioned" data sets that are difficult to deal with. The use of double or multiple precision representations and operations may reduce the error, but not eliminate it.

Algorithms that fail sometimes because of roundoff problems are only one type of algorithm failure. Another not uncommon one is the attempted use of an algorithm to solve a problem other than that for which it is intended. An example of this would be the use of an algorithm designed for the solution of a system of linear simultaneous equations with a symmetric coefficient matrix to solve a system with a nonsymmetric coefficient matrix resulting in an inevitably wrong result.

All too common is the development and use of an algorithm that just will not solve the problem at hand for any set of data, due to a design error, for example, or a failure to understand the underlying mathematics. A vital aspect of the avoidance of such errors is the careful debugging of all newly developed programs using data sets for which the results are known.

Coping with Errors. Since errors are a fact of life in computing, much has been done to assist programmers in locating errors. Syntactic errors are dealt with by the compiler and are not the source of serious difficulty. Although work remains to be done in the area of improving compile-time diagnostics, most compilers provide a reasonably lucid explanation of what has gone wrong. The programmer must then fix the mistake.

Execution-time errors that result from semantic errors are more difficult to deal with. If the program runs to completion, but does not produce the output that is expected, the programmer must carefully examine the output and attempt to locate the fault. The input data should be checked for validity, and then a careful step-by-step analysis of the program must be performed. If the output does not contain sufficient information to determine what the program was doing, then an additional run with detailed print-outs must be made. Special "trace" packages that print out the execution of the program on an instruction-by-instruction basis can be used. Alternatively, only the transfers of control or subprogram references can be printed. If desired, a particular location can be monitored to indicate when the value

was set or referenced. Since the amount of print-out may be voluminous, the programmer must carefully select which features to use. Armed with this material and a thorough understanding of the program, the programmer must perform a careful analysis, which will hopefully lead to the location of the flaw.

If the program does not run to completion but is interrupted as a result of an attempt to perform an illegal instruction, the operating system will (or, at least, should) print a meaningful message. However, since the operating system has no knowledge of what the application program was attempting to do, these messages can be difficult to interpret. Some programming language systems contain an **execution-time** monitor to produce more meaningful diagnostic messages when an abnormal termination occurs.

If a program successfully executes for a given set of data, there is no guarantee that the program will always perform properly. To verify the correctness of a program, multiple sets of test data should be constructed to exercise the **program** as much as possible. As many as possible of the reasonable sets of input data should be run to validate the program. Unfortunately, there are many well-documented cases of programs that have run correctly for many years until a particular set of input data was run and resulted in failure. There is no way to guarantee the correctness of large programs, and programmers must accept the possibility of "bugs" in their programs. Large programs such as operating systems are continuously being modified as faults are located. Perfection in programming is illusory.

The diagnosis of hardware errors has become more complex with the advent of sophisticated hardware architecture constructs such as virtual memory and microprogramming. When it is suggested that a particular error may be a result of malfunctioning hardware, a set of hardware diagnostic programs may be run to assist the maintenance engineer in locating the fault. These programs exercise each of the circuit components and print out the location of the faulty element. This technique is not always successful, since the diagnostic program may not run properly because of the fault. Individual components may have to be removed and tested electrically, or components may be replaced until the machine operates properly.

REFERENCES

1963. Wilkinson, J. H. *Rounding Errors in Algebraic Processes*. Englewood Cliffs, N.J.: Prentice-Hall.
1970. Walker, Terry, and William Cotterman. An

ERRORS, ABSOLUTE AND RELATIVE

Introduction to Computer Science and Algorithmic Processes. Boston: Allyn and Bacon.

1972. Chu, Yaohon. *Computer Organization and Microprogramming*, Englewood Cliffs, N.J.: Prentice-Hall.

A. RALSTON AND B. SHNEIDERMAN

ERRORS, ABSOLUTE AND RELATIVE

For articles on related subjects *see* ERRORS; ERROR ANALYSIS; and NUMERICAL ANALYSIS.

Numerical calculations normally result in an approximation to the true value that is being sought. The *error* in this approximation is a measure of the discrepancy between the true value and the computed result. Estimates of, or bounds on, the error are usually expressed in either *absolute* or *relative* form. Let

TV be the true value of a quantity.

APP be its approximate value.

E be the absolute error.

RE be the relative error.

Then we define

$$E = TV - APP$$

$$RE = (TV - APP)/TV = E/TV$$

For example, if $1/3$ is approximated by 0.333 , then

$$E = 1/3 - 0.333 = \frac{1}{3} \times 10^{-3}$$

and $RE = 10^{-3}$.

More often than not it is E rather than RE that is of interest. For example, if we are calculating the stress on a strut in a bridge, then clearly the absolute error in the calculation is what counts. But in many numerical calculations where the result is very large or very small, relative error is a more meaningful quantity. Thus, if the true value of a quantity is 10^{12} , an error of 10^4 is probably not very serious, but this is more meaningfully expressed by noting that $RE = 10^{-8}$.

Much of numerical analysis is concerned with the estimation of E or RE , or the calculation of bounds on E or RE .

A. RALSTON

EXCEPTION REPORTING

For article on a related subject *see* ADMINISTRATIVE-BUSINESS APPLICATIONS.

Exception reporting is a technique for screening large amounts of computerized data in order to print reports containing only that information requiring action. It differs from the traditional method of reporting, which entails printing the full contents of files and all activity against those files during some period of time. For example, monthly reports analyzing sales data might show the purchases of all regular customers during the month, for this year to date, during the same month last year, and also the year-to-date sales to this point last year. If an organization has 3,000 customers, the report will have 3,000 lines—one for each customer—plus appropriate totals. The manager who receives such a report must review all entries to determine what situations require action. In many organizations, managers are inundated with reams of computer paper presenting too much data to be assimilated and acted upon. This phenomenon is sometimes called “information overload.”

By contrast, exception reports present a user with concise information needed for specific actions. These reports are produced by screening large amounts of information according to predetermined criteria. Exception reports are briefer than conventional reports because they contain only exception data, not all the data in the file. (Fig. 1 illustrates the basic differences between traditional and exception reporting.)

Exception reports generally fall into one of two types, depending on the need they are to meet. Each type is processed according to a different time schedule and uses a different character of screening device. The primary type of exception report is used to isolate exceptions to satisfactory performance. It is produced on a regular, repetitive schedule, and uses a predetermined screening mechanism. Examples of this type of exception reports are monthly inventory reports of all items with balance-on-hand quantities lower than calculated minimums, and monthly and year-to-date sales analyses showing only those customers whose purchases are less than 90% of last year's amounts. (Fig. 1 illustrates this type of report.)

The second type of exception report provides answers to specific inquiries. It is produced only when required, and uses a specially selected screening mechanism. For example, in order to respond to

COMPARATIVE SALES ANALYSIS BY CUSTOMER								
PERIOD ENDING 07/31/74								
PAGE 1								
SLMN NO.	CUST NO.	CUSTOMER NAME	THIS PERIOD THIS YEAR	THIS PERIOD LAST YEAR	YEAR-TO-DATE THIS YEAR	YEAR-TO-DATE LAST YEAR	PRCNT CHNG	
03	4246	HYORO CYCLES INC	3,210.26	4,312.06	10,010.28	9,000.02	11	
	7632	RUPP AQUA CYCLES	7,800.02	2,301.08	20,822.60	16,020.16	29	
	8917	SEA PORT WEST CO	90.00	421.06	900.50	593.10	51	
10	0801	BOLLINGER ASSOCIATES	100.96	0.00	100.06	70.00	44	
	5641	MACK HARDWARE CO	180.60	1,420.96	8,029.22	1,200.00	28-	

COMPARATIVE SALES ANALYSIS BY CUSTOMER								
PERIOD ENDING 07/31/74								
PAGE 47								
SLMN NO.	CUST NO.	CUSTOMER NAME	THIS PERIOD THIS YEAR	THIS PERIOD LAST YEAR	YEAR-TO-DATE THIS YEAR	YEAR-TO-DATE LAST YEAR	PRCNT CHNG	
86	1311	CHESAPEAKE SUPPLY CO	2,551.50	3,126.57	10,914.10	16,424.00	34 -	
	3780	FEDERAL ENGINE CO	1,421.72	1,009.13	8,526.44	6,290.47	36	
98	8850	SOUTH COAST CO	822.59	750.10	3,007.06	4,285.96	30-	

Full Reporting (the traditional method) requires 47 pages of printing

COMPARATIVE SALES ANALYSIS BY CUSTOMER								
PERIOD ENDING 07/31/74								
PAGE 1								
SLMN NO.	CUST NO.	CUSTOMER NAME	THIS PERIOD THIS YEAR	THIS PERIOD LAST YEAR	YEAR-TO-DATE THIS YEAR	YEAR-TO-DATE LAST YEAR	PRCNT CHNG	
10	5641	MACK HARDWARE CO	180.60	1,420.96	8,029.22	11,200.00	28-	

COMPARATIVE SALES ANALYSIS BY CUSTOMER,								
PERIOD ENDING 07/31/74								
PAGE 2								
SLMN NO.	CUST NO.	CUSTOMER NAME	THIS PERIOD THIS YEAR	THIS PERIOD LAST YEAR	YEAR-TO-DATE THIS YEAR	YEAR-TO-DATE LAST YEAR	PRCNT CHNG	
86	1311	CHESAPEAKE SUPPLY CO	2,551.50	3,126.57	10,914.10	16,424.00	34 -	
98	8850	SOUTH COAST CO	822.59	750.10	3,007.06	4,285.96	30-	

Exception Reporting highlights on 2 pages those customers whose purchases this year are significantly less than the previous year

Fig. 1. Sales analysis: full versus exception reporting.

EXCEPTION REPORTING

EXECUTABLE STATEMENT

a one-time government request, a personnel manager may ask for the names of all employees who are eligible for veteran's benefits. The screening mechanism-in this case, all employees eligible for veteran's benefits-is developed just prior to processing, which occurs once in response to this single external request.

Exception reporting appears to offer clear and significant benefits. Why, then, are the vast majority of reports still printed in traditional mode-in lengthly, complete reports?

Both types of exception reports (those produced on a regular schedule and those produced to answer specific inquiries) require screening capability that is available with modern data processing equipment. Yet, many users who have modern equipment are still using systems designed for older machines that did not have the necessary screening capability. Generating exception reports for these application systems requires redesign and reprogramming.

The most common type of exception reports -those produced on a regular schedule-requires the user to have a certain level of knowledge and sophistication. First, a user must be able to determine in advance exactly what information he will need in order to develop the appropriate screening mechanism. Many users have a real or perceived inability to predetermine their data needs. Second, a user must thoroughly understand the report's intended use and the limitations of the report's usefulness. For example, a credit follow-up report designed to list only those customers with account balances more than 90 days outstanding will not provide information about accounts that are overdue but have not reached the 90-day level.

Exception reports can fill a definite need for decision makers who want to make optimal use of the large and growing data files in today's computers. Out of the wealth of data available, only a small amount may be needed for any one decision. Traditional reporting methods provide full data printouts that are useful for general reference, but which contain too much data for timely decision making. Through either regularly scheduled or inquiry reports, exception reporting can provide the concise, focused information a decision maker needs for action.

A. L. TORRANCE

EXECUTABLE STATEMENT

For articles on related subjects see DE-

CLARATIVE STATEMENT; and PROCEDURE-ORIENTED LANGUAGES: Programming.

An executable statement is a procedural step in a higher-level programming language which calls for processing action by the computer, such as performing arithmetic, reading data from an external medium, making a decision, etc. In describing the structure and features of higher-level languages, it is convenient to distinguish between executable statements and nonexecutable or *declarative* statements that provide information about the nature of the data or about the way the processing is to be done without themselves causing any processing action.

Executable statements are sometimes called "imperative" statements because their form often closely resembles that of an imperative sentence in a natural language. For example, the formula

$$Y = a + bx + cx^2$$

follows an imperative form that persists in corresponding structures in programming statements:

[Algol]	$Y := A + B * X + C * X \uparrow 2;$
[Fortran]	$Y = A + B * X + C * X ** 2$
[PL/I]	$Y = A + B * X + C * X ** 2;$

This correspondence is emphasized more explicitly in some languages, namely,

[Basic]	LET Y = A + B * X + C * X ** 2
[Cobol]	COMPUTE Y = A + B * X + C * X ** 2.

Specifications for data transmission between internal storage and an external medium are constructed along similar lines:

[Fortran]	READ (5,12) HERE WRITE (6,21) HERE
[Cobol]	READ INFILE INTO HERE. WRITE OUTFILE FROM HERE.
[PL/I]	READ FILE (INFILE) INTO (HERE); WRITE FILE (OUTFILE) FROM (HERE);

[The numerical specifications in the Fortran example are coded references to additional information about the source (for input), destination (for output), and form of the data to be transmitted.]

Sometimes executable statements are subdivided into imperative and conditional statements because the latter, such as the *IF* statement in Fortran, specify alternate imperative actions linked through a decision mechanism.

A language implementation may have rules about the relative placement of executable and nonexecutable statements. Sometimes it is required that all declarations about data appear before the first executable statement of a program; in other cases it is required only that declarations appear before any information in them is required by an executable statement. One of the distinguishing features of Cobol is its total separation of executable statements (in the "Procedure" division) from non-executable statements (in the "Environment and Data" divisions).

The analogy is sometimes made that executable statements are like the verbs in a natural language, which specify actions, and that nonexecutable statements are like nouns and adjectives, which describe entities and their attributes. The analogy is not complete, of course.

D. D. McCracken and S. V. Pollack

EXTENDED BINARY CODED DECIMAL INTERCHANGE CODE. See EBCDIC.

EXTENSIBLE LANGUAGE

For articles on related subjects see **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.
For article on related term see **ALGOL 68**.

The concept of extensible languages was evolved to permit the user to modify a programming language by adding new features to it or by modifying existing ones. One of the goals was to let the user mold the language to the requirements of his particular area of application, and thus improve his efficiency as a programmer and the clarity of his product.

Extensible languages consist of two basic components :

1. A base language, which provides a complete but minimal set of primitive facilities such as elementary data types, and simple operations and control constructs.

2. Extension mechanisms, which allow the definition of new language features in terms of the base language primitives.

The extension mechanisms can be further subdivided into *semantic extension* facilities and *syntactic extension* facilities.

Semantic extensions introduce new kinds of objects to the languages such as additional data types or operations, whereas syntactic extensions create new notations for existing or user defined mechanisms.

Among others, Algol 68, Basel, EL1 (Extensible Language 1), GPL (General Purpose Language), PPL (Polymorphic Programming Language), and Proteus are languages that are extensible to a higher or lower degree,

As an example of semantic extensibility, consider the mode (i.e., type) and operator definition facilities provided by Algol 68 as demonstrated by the following program segment:

```
mode point = struct (real x, y);
```

Comment: A new object of the mode **point** is being defined as a structure of two real components. The components are accessed by the selectors *x* and *y*. ☐

```
priority -- = 6;
```

Comment: This is declared to be an infix operator symbol of the priority level 6, i.e., the level of addition. ☐

```
op-- = (point p 1, p2) real:
```

```
sqrt ((x of p1 - x of p2) ↑ 2 + (y of p1 - y of p2) ↑ 2);
```

Comment: The symbol --, if applied to operands of the mode *point*, is defined to denote the Euclidean distance of the two points *p1* and *p2*.

The following segment demonstrates how the newly defined objects may be used. ☐

```
begin real a, point u, v
```

```
u := (.0,.0), v := (3.0,4.0); a := u -- v end
```

Comment: *a* is set to 5.0 ☐

It should be noted that all operators so defined are generic, i.e., the same operator symbol may be defined for and used with different operand modes evoking different computations. This is accomplished by checking and matching modes during the compilation.

EXTENSIBLE LANGUAGE

The above example gives a glimpse at the power of the concept. In a similar fashion, these facilities could be used to define polynomials or logical formulas as objects for programs that manipulate formulas, with operations that, for example, add, multiply, intersect, or unite these objects in a formal rather than a numeric way. In computer graphics applications, pictures could be defined as new objects with operators that overlay, scale, or rotate them, etc.

Both mechanisms demonstrated use only notational patterns that are part of the language Algol 68, namely, the mode declaration and the infix operator notation. Algol68 does not allow the user to redefine the syntactic *form* of a statement; thus, it does not provide syntactic extensions.

Where syntactic extension mechanisms are available they usually have the general form

phrase α means β

where α is a syntactic pattern to be defined and β is a program segment consisting of statements of the base language or of extensions previously defined. A GPL program segment will be used for demonstration. In GPL the form

phrase α means β

takes on the appearance

procedure a; β

As in the Algol 68 program above, the following program defines the distance function for points in a plane. It introduces, however, the novel notation

dist a from b

for its invocation. The program assumes that the type *point* has been declared previously:

procedure dist a from b;
iff point a, b take real to be
sqrt ((x(a) - x(b)) \uparrow 2 + (y(a) - y(b)) \uparrow 2);

If this operation is now invoked by, for instance,

length: **= dist x from y;**

x and y must be expressions of the type point; otherwise, a compile-time error message will be issued,

Among the promising properties of extensible languages, the most important is probably that they should make it possible to implement special-purpose languages with relatively little effort. On the other hand, critics say that present-day extensible languages are not powerful enough to accomplish this goal. Some feel that significant alterations of languages are of an inherent complexity too great to be specified simply. Also, there is a difficult trade-off problem between the flexibility that is provided by the language and the efficiency of both its compiler and the object programs generated.

Nevertheless, some degree of semantic extensibility exists in a number of languages (Algol 68, APL, Lisp, Pascal). Languages of this moderate category are sometimes called "enhanceable." Beyond this, it seems that, particularly for special-purpose language development and implementation, some syntactic flexibility is not only desirable but also necessary.

REFERENCES

- 1970. Schuman, S. A., and P. Jorrand. "Definition Mechanisms in Extensible Programming Languages," AFTPS Conference Proceedings, Vol. 37, AFTPS Press, pp. 9-20.
- 1971. "Proceedings of the International Symposium on Extensible Languages" (September 1971, Grenoble, France), SIGPLAN Notices, Vol. 6, No. 12 (December).

J. J. MARTIN

FFT. See **FAST FOURIER TRANSFORM.**

FAST FOURIER TRANSFORM

For article on related subject see **NUMERICAL ANALYSIS.**

The "fast Fourier transform" (FFT) refers to a family of numerical algorithms for computing the discrete Fourier transform (DFT). In complex notation, the DFT is defined by

$$a(n) = \sum_{j=0}^{N-1} x(j) W_N^{nj} \quad n = 0, 1, \dots, N-1 \quad (1)$$

where $x(j)$, $j = 0, 1, \dots, N-1$ is a given sequence of complex numbers and

$$W_N = \exp(2\pi i/N) \quad (2)$$

is the principal N th root of unity. This can be written as a series of sines and cosines by making the substitution

$$W_N^{nj} = \cos(2\pi nj/N) - i \sin(2\pi nj/N). \quad (3)$$

Most of the important applications of the FFT involve the inversion theorem and the convolution theorem.

The inversion theorem states that Eq. (1) is a solution of the system of equations

$$x(j) = \frac{1}{N} \sum_{n=0}^{N-1} a(n) W_N^{nj} \quad (4)$$

This is referred to as the "inverse discrete Fourier transform (IDFT) of $a(n)$."

One important application of a program for computing Eq. (1) is in spectral analysis. Here, one wishes to obtain estimates (with perhaps some smoothing) of the amplitudes and phases given by the $a(n)$ of the sinusoidal components of a signal $x(j)$. Other applications involve the solution of systems of equations by substituting Eq. (4) for the solution and expressing the equations in terms of the $a(n)$. The latter are usually easily solvable and the computation consists mostly of the calculation of the DFT given in Eq. (4).

In some cases, it is expedient to process data by performing operations in the frequency domain, i.e., on the $a(n)$ instead of the $x(i)$. Such applications are usually based upon the convolution theorem, which may be expressed as follows:

Given two periodic sequences $x(j)$, $y(i)$, $j = 0, 1, \dots, N-1$, with DFTs $a(n)$ and $b(n)$, $n = 0, 1, \dots, N-1$, respectively, let the convolution sequence

$$z(j) = \sum_{k=0}^{N-1} x(k)y(j-k), \quad j = 0, 1, \dots, N-1 \quad (5)$$

FAST FOURIER TRANSFORM

have the DFT $c(n)$, $n = 0, 1, \dots, N - 1$. The convolution theorem states that

$$c(n) = a(n)b(n). \quad (6)$$

Therefore, one may obtain $z(j)$ by computing $a(n)$ and $b(n)$ and then computing the IDFT of their product. While the direct computation of Eq. (5) by accumulating products may take a number of operations (multiplications and additions) proportional to N^2 , the use of Eq. (6) will require a number of operations proportional to the time required to compute the DFTs, which is proportional to $N \log N$. Since it is fairly typical to process long records of data in slices having a length of approximately $N = 1,000$, the computation is reduced by a factor of roughly $N/\log_2 N = 100$.

The FFT algorithms use the fact that if N is composite, i.e.,

$$N = r_1 \cdot r_2 \cdot \dots \cdot r_m, \quad (7)$$

the series (1) can be expressed as a nested sequence of series of subseries which requires a number of operations proportional to

$$N_{op} = N(r_1 + r_2 + \dots + r_m). \quad (8)$$

For $N > 4$, this is less than the N^2 operations that would be required by a direct accumulation of products for each value of j according to the defining formula (1). The number N_{op} is minimized by using as many factors as possible. If all factors are equal to r , then

$$N_{op} = Nr \log_2 N = \frac{r}{\log_2 r} N \log_2 N. \quad (9)$$

A frequent choice, for programming efficiency, is to select N to be a power of 2 so that

$$N_{op} = 2 \cdot N \log_2 N. \quad (10)$$

The algorithm is easily derived when N is a product of two factors, r_1 and r_2 . The indices n and j in Eqs. (1) and (4) are, in this case, replaced by index pairs (n_1, n_2) and (j_1, j_2) , respectively, defined as

$$\begin{aligned} n &= n_1 + r_1 n_2, \\ j &= j_1 + r_1 j_2, \end{aligned} \quad (11)$$

where

$$\begin{aligned} n_1 &= 0, 1, \dots, r_1 - 1, \\ n_2 &= 0, 1, \dots, r_2 - 1, \\ j_1 &= 0, 1, \dots, r_1 - 1, \\ j_2 &= 0, 1, \dots, r_2 - 1. \end{aligned} \quad (12)$$

Using Eq. (11), one can perform the factorization

$$W_N^{-nj} = W_N^{-n_2 j_1 r_2} W_N^{-n_1 j_1 r_2} W_N^{-n_2 j_2 r_1} W_N^{-n_1 j_2}, \quad (13)$$

which is easily simplified by using the relations

$$W_N^{r_1 r_2} = 1, \quad W_N^{r_1} = W_{r_2}, \quad W_N^{r_2} = W_{r_1}. \quad (14)$$

The summation may then be taken over j_1 and j_2 instead of j to give

$$\begin{aligned} a(n_1 + r_1 n_2) &= \sum_{j_2=0}^{r_2-1} \left\{ \sum_{j_1=0}^{r_1-1} x(j_2 + j_1 r_2) W_{r_1}^{-n_1 j_1} \right\} \\ &\times W_N^{-n_2 j_2} W_{r_2}^{-n_2 j_2} \end{aligned} \quad (15)$$

The inner sum is, for each of the r_2 values of j_2 , a DFT of an r_1 -point sequence which, for all r_1 values of n_1 , can be computed in r_1^2 operations. After multiplication of this result by the phase factor $W_N^{-n_1 j_2}$, or "twiddle factor," the outer sum may be calculated as a DFT of an r_2 -point sequence, which for each of the r_1 values of n_1 takes r_2^2 operations. If the phase factor $W_N^{-n_1 j_2}$ is absorbed in either of the W_{r_1} or W_{r_2} factors, this will take a total of

$$N_{op} = r_2 r_1^2 + r_1 r_2^2 = N(r_1 + r_2) \quad (16)$$

operations. If r_2 can be factored into $r_2' \cdot r_3'$ and the process repeated on the r_1 -point DFT, it is easily seen that the number of operations will be

$$N_{op} = N(r_1 + r_2' + r_3'). \quad (17)$$

By an inductive argument, one arrives at Eq. (8).

The algorithm described by Eq. (15) may be written so that the data is overwritten by the results, with practically no other storage required except, perhaps, that used for a table of sines.

REFERENCES

- 1976, Cooley, J. W., P. A. W. Lewis, P. D. Welch. **Practical Fourier Analysis, the FFT and Its Applications**. New York: John Wiley.
- In process.----. "The Fast Fourier Transform and Its Application to Time Series Analysis," in Enslein, Ralston, and Wilf (Eds.), **Mathematical Methods for Digital Computers**, vol. 3. New York: John Wiley.

J. W. COOLEY

FEASIBILITY STUDY

For article on related subject, see **DOCUMENTATION**.

The feasibility study takes place at the beginning of a systems development project and leads to a *feasibility report*. It is a broad-brush study which seeks to determine two things: the exact nature of the problem to be solved and an outline of one or more solutions to the problem. It seeks to answer three questions: technical-will it work?; economic-will it pay?; operational/political-will it be used? The feasibility report is submitted to the problem proponent (user) who, after review, may authorize the detailed development work on a selected solution, modify the design, or possibly abandon the whole project.

The basis of the feasibility study is that it enables fundamental decisions to be made before time and money are committed to the systems and programming work. Provided the feasibility study is conducted carefully by experienced personnel, the outline solution(s) and cost/timing estimates will be confirmed in the detailed systems work. But, in practice, the findings of the study and the decisions based upon it may be reviewed in the light of the detailed investigation and analysis, and the project modified (or even abandoned) any time up to the beginning of programming. Revisions to the scope of the project or the design approach after programming has begun will be very expensive indeed. An example of the contents of a feasibility study report is given in Fig. 1.

As Fig. 1 shows, the study is based on an investigation of the proponent's (user's) problem area. The scope of the investigation and the resources required will depend on the type of project. The feasibility study for the development of a major accounting or production control system in a large company may take several systems analysts a number of months to complete. A study of a small "subsystem" to produce an additional report from an existing system, on the other hand, may take only a few days.

The statement of the user's requirements generally includes the following categories:

1. *Objectives*. States what the user wants the new system to achieve. This should be quantified wherever possible. For example, rather than a general statement of "increase factory throughput," a quantified statement of requirements is preferable, such

as "to reduce work in progress by 15% and to increase machine utilization by 5%."

2. *Boundaries and Constraints*. Stipulates what the user does not want changed or sets forth any restriction on the design approach or the facilities used. These may be imposed for various technical and business reasons. Examples: "The job docket currently being used should not be changed." "The system must not be justified on staff displacement." "No more clerks in Grades A and B can be used." "Existing computer facilities must be used."

3. *Time Scale*. Sets the date when the new system is required. There may be many good business reasons for requiring a new system to be operational by a certain date. For example, the new system must be available by the end of the financial year, or before research begins on a new project, or to coincide with next annual stock inventory.

4. *Mandatory Reports*. Lists mandatory output reports, identified by the user, which must be produced in the new system.

This problem definition is the design brief for a systems analyst. His task is to produce and outline one or more solutions that meet the user's needs as described above. A range of solutions is generally preferable because this will give the user an objective choice (rather than "forcing" one solution upon him). For example, if there is an existing system, one alternative solution is "do nothing-do not change method of working"; this will serve as a basis for comparison with new system designs. For each solution there must be sufficient information given to enable the user to answer the following questions:

"Does the system meet my requirements?"

"How much will it cost and how long will it take?"

"Is it worthwhile to proceed?" (a cost benefit analysis of expenditure against savings)

This means that a description of the proposed new system must be given, together with a statement of the objectives that are met by the design. The benefits can be estimated from the latter. Some benefits will be tangible and quantified, others will be intangible and qualified benefits on which no monetary value can be placed. From the outline design, estimates can be made of the resources and time necessary to develop the system; the operational costs of running the system can also be assessed. The cost/benefit analysis, using the appropriate costing methods, can then be drawn up.

K. R. LONDON

FEASIBILITY STUDY

	FEASIBILITY STUDY REPORT
	Title
	Contents
	Administrative Information
1.	INTRODUCTION
	1.1 Terms of Reference
	1.2 Method
	1.3 Summary
2.	FINDINGS
	2.1 Description of Existing System
	2.2 Identified Problems
	2.3 User Request
	-Objectives
	-Boundaries and Constraints
	-Timescale
	-Mandatory Reports
	-Assumptions and Limitations
3.	SYSTEM SOLUTION (repeated for each solution)
	3.1 System Outline
	3.2 Benefits Realized
	3.3 Development Schedule, Resources and costs
	3.4 Operating Schedule, Resources and costs
	3.5 Advantages and Disadvantages
	-Cost/Benefit Analysis
	-Impact on Existing Organization/ System
	-General Comments and Summary
4.	RECOMMENDATIONS AND CONCLUSIONS
	4.1 Conclusions (these may include a comparison of solutions)
	4.2 Recommendations
	4.3 Further Actions
	APPENDICES
	(Detailed timing and cost figures, document samples, etc.)

Fig. 1. Sample feasibility study, table of contents.

FIFO-LIFO

For articles on related subjects see **DATA STRUCTURES**; and **STACK**.

For articles on related terms see **LIST AND LIST PROCESSING**; and **QUEUEING THEORY**.

The terms FIFO and LIFO refer to two techniques for dealing with collection of items to which additions and deletions are to be made. The acronym FIFO stands for first-in-first-out and LIFO represents last-in-first-out. Derived from business accounting and inventory management notions, these techniques have found widespread application in computer science.

The FIFO concept is based on the simple idea of people waiting on line to be serviced at a bank teller's window, a supermarket checkout counter, or a bus stop. The first person to arrive is serviced, and if there is a line of customers the order of entry to the rear of the line is the order of service given at the front of the line. The same concept can be applied to ships waiting to unload at a dock, to jobs waiting to be run in a computer system, or to airplanes waiting to be serviced by a repair shop. The line of people or items waiting to be serviced is called the "queue" (Fig. 1).

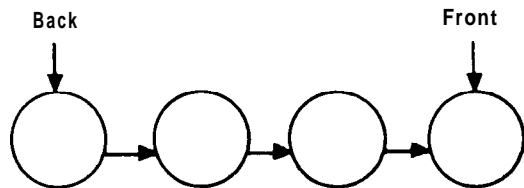


Fig. 1. FIFO queue; additions at back; deletions at front.

There are a great number of variations to the basic theme of FIFO arrangement. Multiple-server queues have a single queue, but several facilities that provide service. Some banks and airline ticket counters have adopted this technique by having a single line that feeds to a group of teller windows. Priority queueing permits persons with high priority to move up to the front of the queue. **Bounded-length** queueing puts an upper limit to the number of persons in the queue.

The LIFO concept is based on the notion that the most recently arrived item is dispatched first. Thus, the freshest vegetables in the grocery are sold first and the inventory items most recently put on the shelf are the first to be sold. This idea is familiar to

card players in some games (gin rummy, for example), who may take a card from the top of the pile or place another card face up on the pile. The stack of plates on the spring-loaded dispenser found in cafeterias is another common example of the LIFO principle. The usual definition of this principle includes the specification that only the top element of the collection may be removed and that new items may be placed only on the top of the collection. A collection that has these rules for addition and deletion is called a "stack" or a "pushdown list" (Fig. 2). Automata theorists distinguish between these two terms: In a stack, the interior items may be examined; in a pushdown list, the interior items may not be examined.

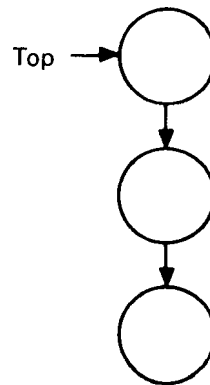


Fig. 2. LIFO stack: additions and deletions at top.

The LIFO technique has widespread application in computer science, particularly in the parsing techniques employed by compilers and in the searching of data structures,

B. SHNEIDERMAN

FILES

For articles on related subjects see **DATA BASE AND DATA BASE MANAGEMENT**; **RECORD**; and **STORAGE MANAGEMENT STRUCTURES**.

Background. The term "file" must have been one of the first to be used in commercial data processing terminology. Even before the advent of computers a deck of punched cards was often called a "card file," a term also applied to the cabinet in

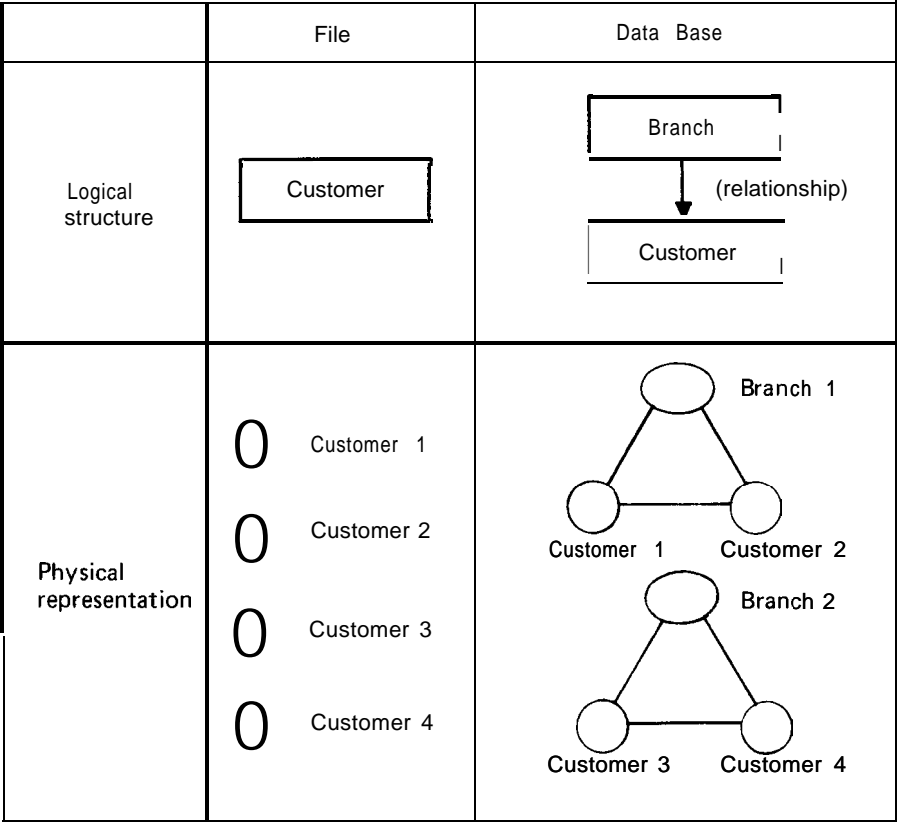


Fig. 1. Files and data bases.

which the cards were stored. In the very early days of computers, any collection of data or programs was identified as a file, Magnetic tape was the first storage medium on which data could be stored, detached, and then attached again at a later date. This led to the widespread use of the term “tape file.”

When removable disk packs came into use as a storage medium, the concept of a file became slightly more complex. Before the era of data processing, a file was traditionally an ordered collection of similar entities. Magnetic tapes did nothing to disrupt this simple view, but disks and other direct-access devices had an extra dimension. Any record in the file could be accessed without looking first at those preceding it in the order. In this way the concept of an indexed file came into being. Many ways of accessing the records in the file were developed, and each way became known as an “access method.”

Definition. A file is a collection of data records (possibly of different types) in which the

records have no relationship other than that they are in the same file. Some confusion in this area has resulted from the introduction in the mid-1960s of the term “data base.” The concept of a data base has become fashionable, and sometimes collections of data which previously had been happily referred to as files received the accolade of “data base.” We distinguish these two terms by defining a data base as a collection of data records (of two or more types) in which relationships exist and are represented between record types and between records of the same type.

Fig. 1 illustrates the logical structure and physical representation of a file and a data base.

Kinds of Files. The word “file” is used in many ways in data processing. Examples are data file, input file, output file, master file, program file, working file, scratch file, temporary file, transaction file, operating systems file, log file, print file, card file, and job file.

The preceding definition applies essentially to a data file, i.e., a file in which a program may operate during the course of its execution. However, the data may in fact consist of programs, with each program statement as a record in the file.

T. W. OLLE

FINITE ELEMENT METHOD

For articles on related subjects see **ENGINEERING APPLICATIONS; MATRIX COMPUTATIONS; NUMERICAL ANALYSIS; and PARTIAL DIFFERENTIAL EQUATIONS, NUMERICAL SOLUTION OF.**

The finite element method is a relatively recent and very powerful approximate technique used to solve field problems in various engineering fields. Its development follows closely the increasing usage and availability of large-scale digital computers. Some areas of practical application are:

1. Static and dynamic analysis of complex structures such as airplanes, bridges, buildings, dams, ships, and cars.
2. Fluid flow, diffusion, and consolidation problems.
3. Liquid sloshing in an elastic container.
4. Lubrication problems.
5. Heat conduction and thermal stresses.

In the past decade, through applications of the technique, many engineers, mathematicians, and computer scientists have gained a large amount of direct experience with this new concept. We will briefly discuss this method as seen from the viewpoints of each of these three professional groups.

The Practicing Engineer. Practically on the basis of physical intuition alone, this method was initially used for aerospace structural analysis. Essentially, a continuous system is idealized as an assembly of discrete elements (Kuo, 1961). For example, an automobile may be idealized as a set of triangular elements as shown in Fig. 1. The common shape of a discrete element, known as the "shape" function, is the triangle for a two-dimensional structure and the tetrahedron for a three-dimensional one. However, other shape functions have also been used (see Fig. 2). Once this decision is made, the

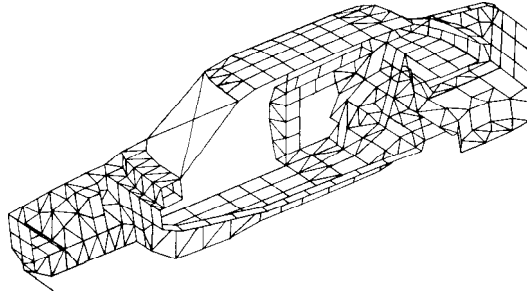


Fig. 1. Idealization of a car body using finite elements.

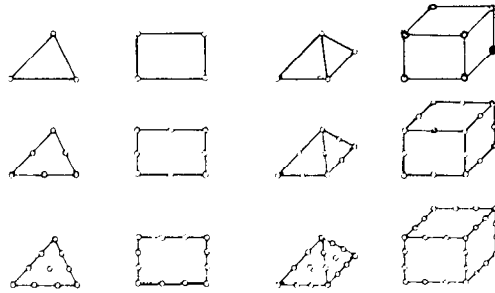


Fig. 2. Some common two-dimensional and three-dimensional finite elements.

remaining steps follow a standard procedure. For example, in a structural problem using a stiffness matrix (Desai and Abel, 1972; Zienkiewicz, 1967), they are:

1. Choice of the displacement function for each element, assumed to be a polynomial usually, in terms of displacements at the nodes.
2. Selection of the stress-strain relationship.
3. Development of the stiffness matrix k for an element.
4. Consolidation of internal degrees of freedom.
5. Formation of k for the assemblage.
6. Computation of the displacements, strains, and stresses at various nodal points.

The Applied Mathematician. Many applied mathematicians view the finite element method as the approximation of a continuum by elements, each with multiple connecting points. It is similar to the Rayleigh-Ritz method with the following differences: (1) The piecewise continuous field definitions used in the finite element method are used to take care of irregular boundaries; and, (2) the resul-

FIX

tant equations from the finite element method normally consist of banded or sparse matrices.

Mathematicians are also quick to point out that, while the finite difference method involves mathematical lumping, the finite element method uses physical lumping. Moreover, the finite element method, using a triangular element, is equivalent to the "hypercircle" method developed in 1943 by R. Courant. Today, various variational forms of the finite element method are being investigated (Pian and Tong, 1972), and a finite element approach to the solution of partial differential equations is being extensively studied (Aziz, 1972).

The Practicing Computer Scientist. In practice, the matrix formulation seems to be the most efficient way to organize the finite element solution for an idealized discrete model, but in adopting it, the computer scientist faces various new and challenging problems. They include (Oden et al., 1972):

1. Large-scale matrix problems involving banded or sparse matrices. The associated error bounds and convergence must be studied. Overlay techniques are often used.

2. Automatic mesh generation to avoid the error-prone input of large amounts of data. A typical mesh pattern (automatically generated) is shown in Fig. 3.

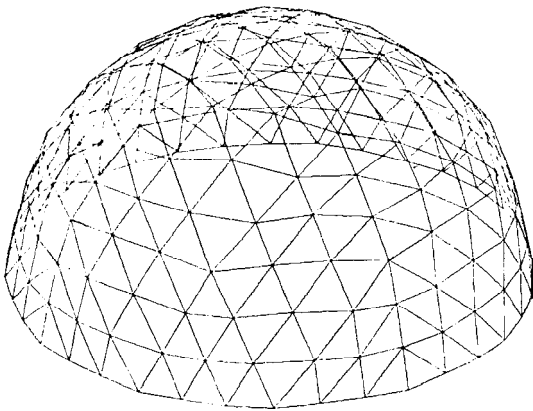


Fig. 3. Computer-generated mesh pattern.

3. Computer-oriented interpolation techniques for two-dimensional and three-dimensional problems.

4. Hidden-line computer graphics for output.

5. Design of a general-purpose computer program versus some specially designed programs.

REFERENCES

1961. Kuo, S. S. "On Jacobi's Method for Real Symmetric Matrices," *Journal Of the Aerospace Sciences*, Vol. 28, No. 3, March, p. 255.
1967. Zienkiewicz, O. C. *The Finite Element Method in Structural and Continuum Mechanics*, New York: McGraw-Hill.
1972. Aziz, A. K. *The Mathematical Foundations of the Finite Element Method with Applications to Partial Differential Equations*. New York: Academic Press.
1972. Desai, C. S., and J. F. Abel. *Introduction to the Finite Element Method*. New York: Van Nostrand-Reinhold.
1972. Oden, J. T., R. W. Clough, and Y. Yamamoto (Eds.). "Advances in Computational Methods in Structural Mechanics and Design," Section IV. Huntsville: Program Development, University of Alabama Press.
1972. Pian, T. H. H., and P. Tong. "Finite Element Methods in Continuum Mechanics." In Chia-Shun Yih (Ed.), *Advances in Applied Mechanics*, Vol. 12. New York: Academic Press.

S. s. KUO

FIX

For article on related subject see **PATCH**.

For article on related term see **SOURCE PROGRAM**.

As a verb, the word "fix" means to patch a program in an attempt to correct a bug (an error). More often it is used in computing as a noun. The fix can be made at the machine language level, but commonly, in assembly language or higher-level language coding, the source code is corrected and the entire program is reassembled or recompiled.

F. GRUENBERGER

FLOW DIAGRAM

For articles on related subjects see **BLOCK DIAGRAM**; **DOCUMENTATION**; **FLOWCHART**; and **SYSTEM CHART**.

FLOWCHART

A flow diagram is a variety of flowchart. The flow diagram is distinguished from other varieties of flowchart by its emphasis upon the algorithm. The flow diagram depicts the details of how an algorithm is to transform data when used as a representation for a computer program. Flow diagrams can be distinguished from other varieties of flowchart by the relatively infrequent presentation of input/output operations. An example of a flow diagram is given in Fig. 1.

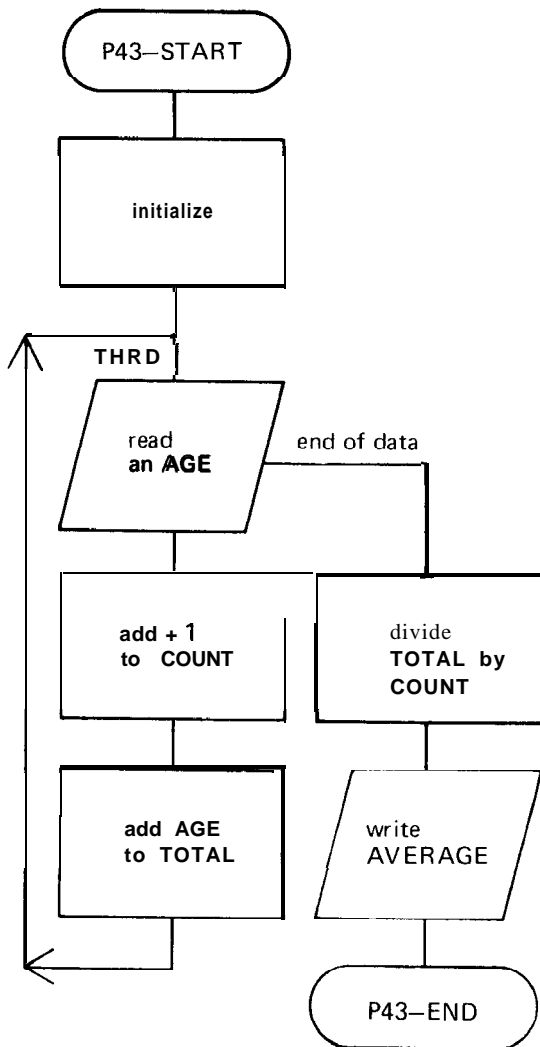


Fig. 1. Example of a flow diagram for finding an average, (From Ned Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

N. CHAPIN

For articles on related subjects see **ALGORITHM**; **BLOCK DIAGRAM**; **DOCUMENTATION**; **FLOW DIAGRAM**; **PROGRAM**; and **SYSTEM &ART**.

For article on related term see **STANDARDS**.

Definition. Flowcharts are graphic means of describing a sequence of operations done on data. They serve as a means of communication from one person to another about transformations on data. Flowcharts are graphic because they commonly use a two-dimensional pictorial format. The placement of the identifications of the operations in the pictorial format shows the sequence of the operations. The pictorial format typically incorporates wording to identify the data and the operations. The pictorial format is the subject of both an International and an American National Standard (ANSI, 1970).

Flowcharts get their name from their chart (graphic) representation of the flow (orderly passing of control) from one operation to the next in an explicit sequence. Flowcharts go by many other names, including block diagram, flow diagram, logic diagram, system chart, run diagram, process chart, procedure chart, and logic chart (Chapin, 1971). These different names reflect in part a lack of uniformity of nomenclature and in part the particular interests of specialized users. For example, prior to the advent of computers the name "flowchart" was used by systems analysts to designate a means of describing the flow of documents carrying data in an organization.

The two major varieties of flowchart in present-day practice are the system chart and the flow diagram. As indicated in Fig. 1, the flow diagram concentrates on part of what a system chart shows. The unit of data transformation for the two is thus very different. For a flow diagram, the unit of data transformation is usually an operation or short sequence of operations that a computer can perform (such as an instruction or a series of instructions that comprise a subroutine). An example is testing for the presence of leading zeros in a number,

By contrast, in a system chart the unit of data transformation is usually the work done by an entire computer program. Examples are: sorting a file of data, inverting a matrix, or producing a report. Flow diagrams commonly have an algorithmic orientation, stressing how data is transformed, whereas system charts primarily identify inputs and outputs to algorithms, stressing what data is used or pro-

FLOWCHART

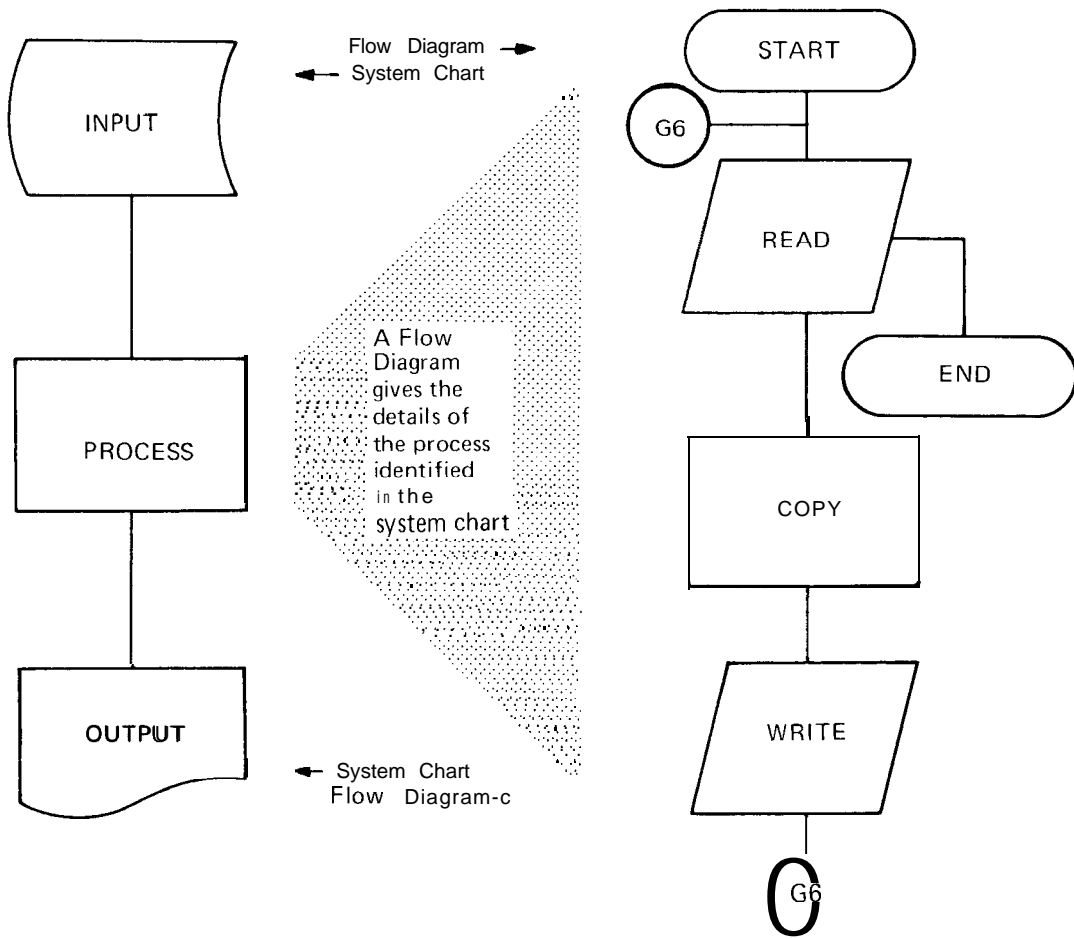


Fig. 1. Two types of flowcharts: the system chart and the flow diagram. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

duced at various points in a sequence of operations. System charts and flow diagrams are described in more detail later in this article.

The major varieties of flowcharts are alike in that they describe operations in sequence on data. They stress "what" and "how." Flowcharts are weak on "when," "why," and "what does the action." The only usual indication of "when" is the relative position in a sequence of operations. Usually, the doer of the action is the computer arithmetic and logic unit for a flow diagram, and the computer itself for a system chart, with exceptions possible.

Use. Flowcharts are the most widely used graphic method for describing computer operations. They are adaptable to a wide variety of different

applications and circumstances and have enjoyed use from the early days of the computer field (see Chapin, 1971, for a brief history).

The major use of flowcharts is in documentation and in programming. As a documentation device, the flowchart provides a way of communicating, from one person to another, the nature of the operation to be performed and of the data upon which it is to be performed, regardless of the programming language or computer used. Since a flowchart is a graphic means of communication, this feature makes it a good choice for use with the usual programming language and English language descriptions included in most documentation. This alternative means of description can enhance the usefulness of the other means.

FLOWCHART

As a programming aid, flowcharts are often prepared by systems analysts and designers to describe systems and to specify the work to be accomplished by programs. Programmers use flowcharts as a basis for writing programs and as a means of communicating among each other, particularly when the programming is done as a team effort. Programmers as well as systems analysts also use flowcharts as a source of information for maintenance work on programs and systems.

Flowcharts are more widely used than decision tables, publication languages, or abstract notations (such as the Iverson notation used in API., or the lambda calculus). This popularity appears to be due to the balance of advantages and disadvantages.

ADVANTAGES. Flowcharts have few features limiting their use. Hence, they are broadly applicable across a wide range of industries, computer applications, and types of work. For example, they are as convenient in administrative file handling as in scientific or engineering computation. This popularity, bred of wide applicability, generates further use as a *lingua franca* among persons working with computers.

Flowcharts are also largely language independent. Knowledge of a programming language is not normally necessary to be able to use them or to create them. This is always true for the pictorial part of flowcharts, and ideally should be true for the wording or symbols incorporated in a flowchart.

Flowcharts are constraining and precise in ways that are useful to programmers and analysts. They are a limited means of description that force the user to give attention to many significant matters while suppressing attention to a host of less important matters.

Flowcharts are a visual representation, and hence provide a convenient alternative to the usual narrative description for a program or system. This enables a more rapid scan or search of a flowchart than of a narrative description when particular items of information are sought. The graphic format enables a user to comprehend much at a glance.

Flowcharts offer a controllable level of detail. They are usable from the most summary systems level to the most detailed programming level, at the option of the one who prepares the flowchart. This wide range of detail options greatly enhances the communication value of the flowchart. It is generally conceded that the flowchart is most valuable to the user when the level of detail in the flowchart is more summary than that provided in the programming language (for flow diagrams) or in the English language narrative (for system charts). For this

reason, the person who creates flowcharts commonly prepares them in a series, arranged from very summary to quite detailed, so that the user may choose the level of detail most convenient for his particular purposes as they change from time to time. In this regard it should be noted that the system chart variety of flowchart inherently provides a more summary view than does the flow-diagram variety of flowchart.

DISADVANTAGES. On the disadvantage side, some programmers and analysts complain that flowcharts are a waste of time, since people do not think in graphic terms. In their view, a flowchart is therefore an unnatural means of communication.

Flowcharts are often cumbersome to use and costly to produce. Because of their graphic format, flowcharts may devote more than a page of space to present what may require from a few lines to less

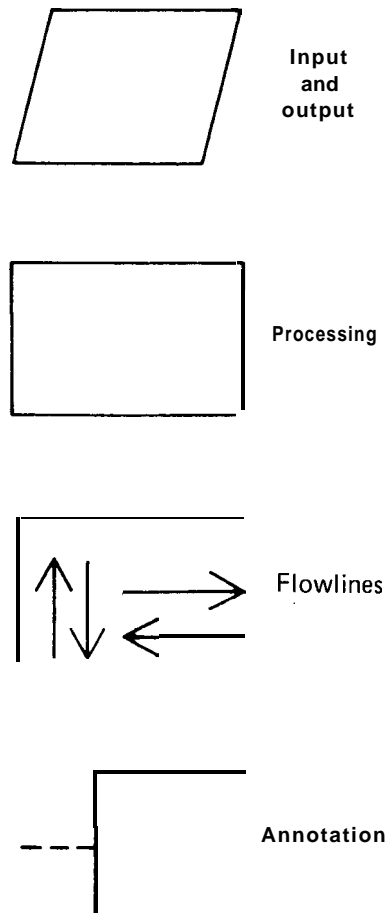
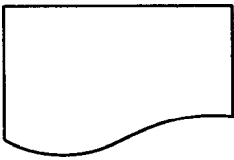


Fig. 2. Basic outlines. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 197 1.)

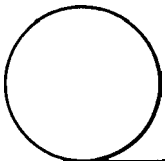
FLOWCHART



Document



Punched
card

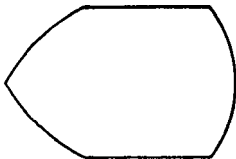


Magnetic
tape

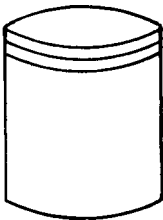


Punched
tape

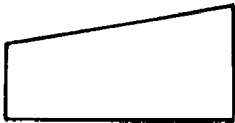
(a)



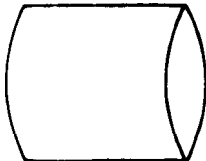
Di splay



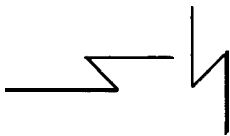
Disk
storage



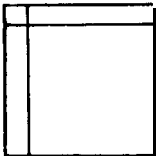
Manual
input



Drum
storage



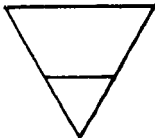
Communication
link



Core
storage

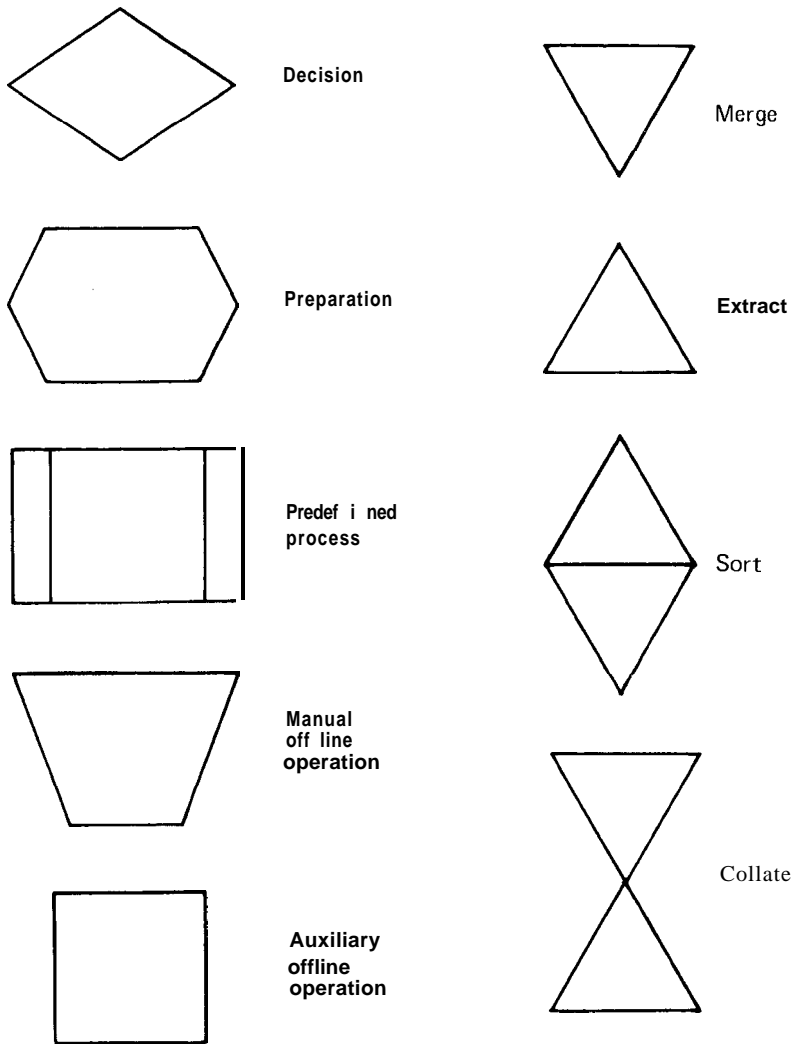


Online
storage



Off line
storage

(b)



(c)

Fig. 3. Specialized outlines. (a) Media. (b) Equipment, (c) Processes. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

FLOWCHART

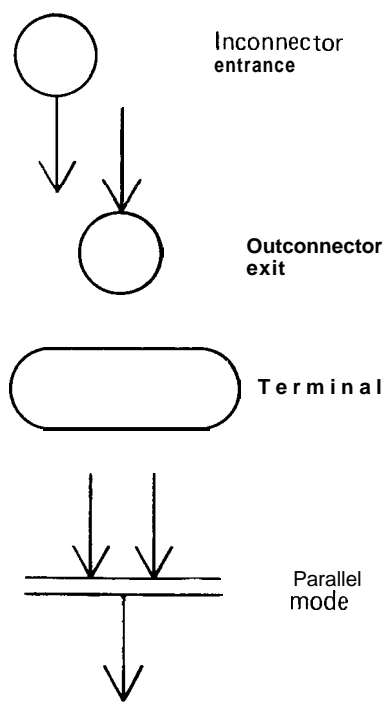


Fig. 4. Additional outlines. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

than a half-page of equally detailed description in some other form. Manual preparation of flowcharts is slow, although detailed flow diagrams can be prepared by computer if the program to be flow-charted exists in source language form.

Flowcharts do not constitute a programming language. They are person-to-person means of communication, not person-to-computer. No translators exist for accepting programs or systems described in flowchart form.

The flow diagram variety of flowchart does not fit well with all programming languages. Although flow diagrams go well with Cobol, Fortran, Algol, PL/I, and Basic, for example, they seem ponderous or incongruent with Snobol, Comit, Lisp, or IPL-V.

Flowcharts may not highlight what is important. Each operation commonly receives as much attention in a flowchart as any other, given the level of detail at which the flowchart is prepared. Yet, intuitively, people feel that some operations are more significant than others. The flowchart does not provide any convenient, automatic way to highlight these.

Flowcharts are difficult to produce at a summary level. No consistent logical rules have yet been developed to aid the process of producing meaningful summary flowcharts at a consistent level of detail. When many operations are to be condensed

\vdash	is replaced by	:	comparison	—	underline, blank
\leftarrow		>	is greater than		Absolute value
+	plus or addition	=	is equal to	\neg	negation
	minus or subtraction	<	is less than		logical OR
*	multiplication	\nless	is not greater than	&	logical AND
x		\rightarrow		,	literal (zero level address)
**	exponentiation	\neq	is not equal to	()	grouping, level of address
↑		$\neg =$		A ()	address constant
/	division	\nless	is not less than		
EOF	end of file				

Fig. 5. Symbols for use in flow diagrams. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

and summarized into one operation, serious problems arise. What should be the summary operation? What details are to be suppressed?

Elements. Wide agreement exists on the major elements of flowcharts. They are the outlines, flow representation (sequence), and the symbols specifying the operation and identifying the data. Each interacts with and augments the work of the others, to provide the graphic presentation.

The outlines, sometimes called "symbols," but called "boxes" in common speech, are in three groups: the basic, the specialized, and the additional. Complete flowcharts can be drawn using only the basic outlines. These outlines, as shown in Fig. 2, are a parallelogram for input or output, a rectangle for processing, and a line or lines with open arrowheads to represent the direction of flow, as commented upon later. When only the basic outlines are used, the rectangle serves for all operations except input and output. To provide explanatory data, the annotation outline (a partial rectangle attached by a dashed line) can be used, with the comment written within the rectangular part.

Specialized outlines are numerous, as shown in Fig. 3a. These offer a way of visually identifying for the user of the flowchart the media used to carry data; the equipment used for input (Fig. 3b), output, or storage of data; and the general character of a processing operation, such as data transformation (Fig. 3c).

In system charts, the most commonly used of these specialized outlines are usually the document, magnetic tape, on-line storage (for data on magnetic disk, commonly), and punched card outlines. In flow diagrams, the most commonly used are the preparation, the decision, and the predefined process outlines. A predefined process is considered to be a sequence of operations not described in the flowchart but incorporated in the program, as by a call to a subroutine in a library. The manual operation outline is for representing operations done by people, such as "review data for conformance to policy." The auxiliary operation outline is for representing operations done by noncomputer machines, such as "interpret punched cards."

The additional outlines, as shown in Fig. 4, are connectors of three types, and serve to indicate that two or more sequences are to be performed simultaneously (parallel mode). The in-connector and the out-connector are distinguished by the direction of the flowlines leading to and from them. An in-connector, or entrance connector, has a **flowline** leading from it, and is placed to the left or above the

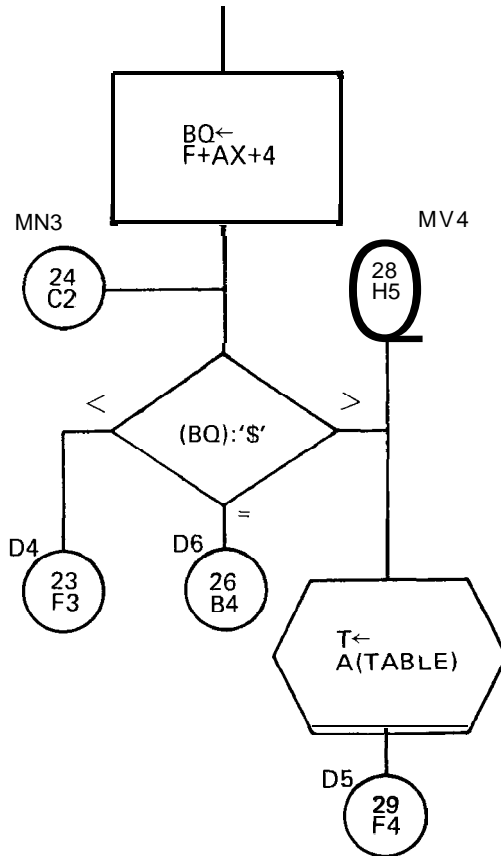


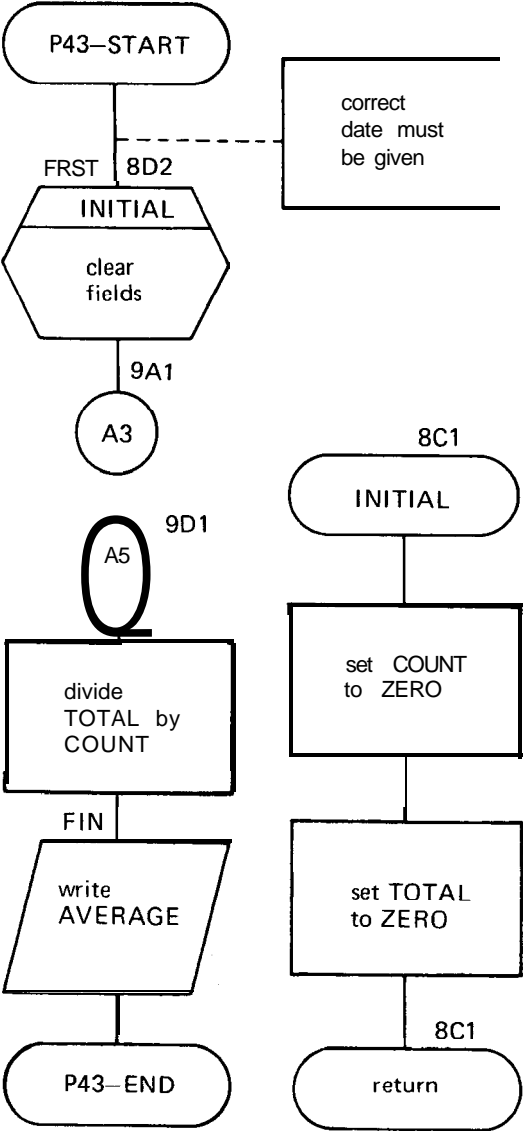
Fig. 6. Entry and exit flowlines in a flow diagram. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

main line of flow or sequence. The out-connector, or exit connector, has a **flowline** leading into it, and is placed to the right or below the main line of flow or sequence. A terminal connector may also be used in an entrance or exit position as a marker to indicate the beginning or ending of a sequence of operations.

The wording within process outlines identifies the operation and (in flow diagrams) the data involved. The wording within the input or output outlines identifies the data and (in flow diagrams) whether the data is input or output. The wording used within and with the outlines in a flowchart has not been standardized, although the problem has been studied (ANSI, 1965). Common practice depends in major part upon the level of detail to be depicted. In summary flowcharts, English language phrases are the most common. Detailed system charts often include only the names of the inputs, outputs, and programs. Very detailed flow diagrams

FLOWCHART

Page 8:



Page 9:

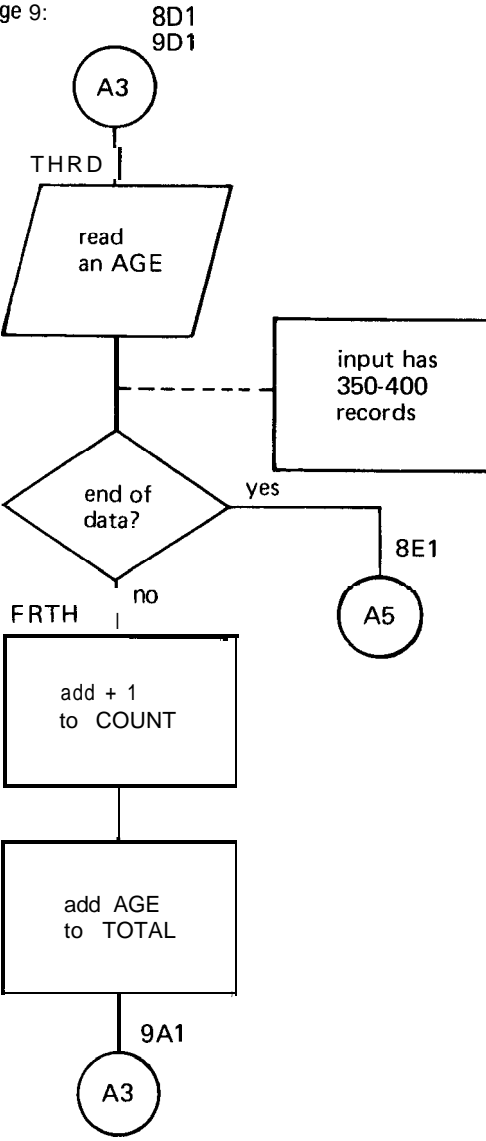


Fig. 7. Example of cross referencing in a flow diagram. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

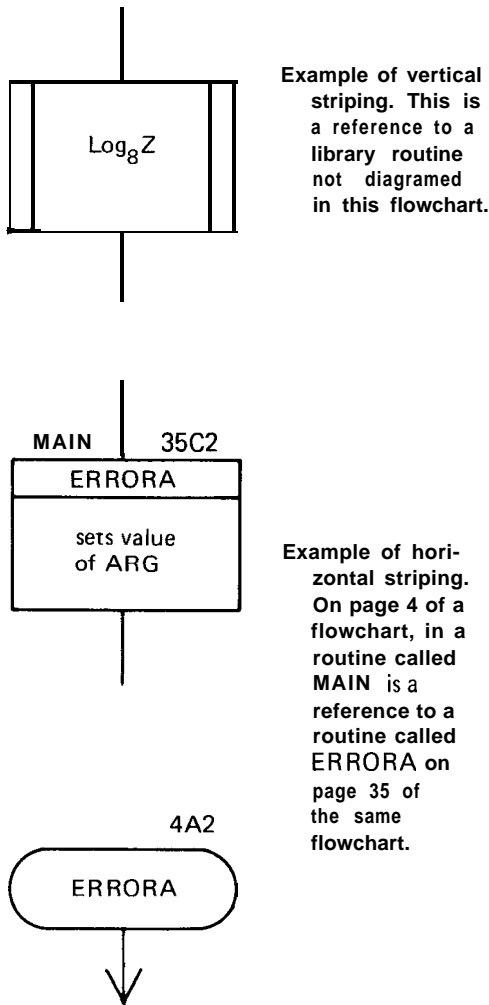


Fig. 8. Conventions for striping and references.
(From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 197 1.)

often use programming language instructions within the outlines; less detailed ones use symbols for operations, borrowed largely from logic and mathematics (shown in Fig. 5) or English language words.

Conventions. To preserve the graphic format and to improve the usefulness of the flowchart in conjunction with program listings and written system specifications, programmers and analysts use a number of conventions in preparing flowcharts. Some of these conventions apply to ways of depicting convergent and divergent flows, some to the

substitution of one outline to represent many, and some to cross-references.

The flow direction (sequence of operations represented) is from top to bottom and left to right. Most deviations from this rule are marked by arrowheads, with the barbs showing the flow direction. Broken flows are marked by connectors in flow diagrams. An exit connector cuts off a flow; an entry connector resumes a flow, as shown in Fig. 6. In system charts, the usual practice is to repeat as input outlines, when resuming a flow, the output outlines that existed when the flow was cut off. Breaks in flows are common at the edges of pages, but may occur anywhere in a flowchart. To facilitate finding the other end of a break in flow, the common practice is to annotate both ends of the break with location cross-references to the other ends, as shown in Fig. 7.

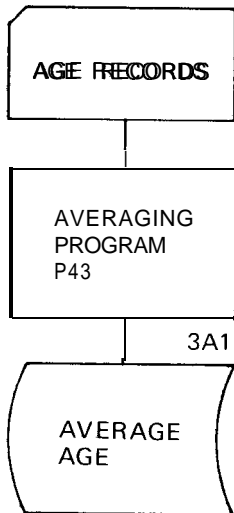
Convergent flows are shown by entrance connectors to create physically joining flowlines, together with location cross-references. Divergent flows are handled differently. In system charts they are represented by just the reverse of the convention used for convergent flows. In flow diagrams, the convention is more complex; divergent flows must come from a decision process. It may be implied, as when an end-of-data condition is encountered in attempting an input operation, but usually it is explicit, based on a comparison process. Each multiple exit flowline from a decision operation must be identified as to the basis for selecting it (see Fig. 6). If a break in flow is also involved, then a location cross-reference is also commonly added, as illustrated in Fig. 7.

At any point in the flowchart, the person who prepares it may add cross-references to other materials, such as system descriptions or program listings. This is usually done by writing in identifying names ("entry points" or "labels") adjacent to a process or connector outline, as shown in Figs. 6 and 7. These names are taken from the corresponding points in the materials referenced. Additional or more extended cross-referencing can be done by using the annotation outline.

To save space in parts of flowcharts in order to provide for a more meaningful presentation of the operations, the horizontal striping convention is an aid. This allows one outline, suitably identified, to represent a sequence of outlines presented elsewhere in the flowchart, as shown in Fig. 8. Most commonly, it applies to process outlines. The person who prepares the flowchart draws a horizontal line through the outline, and identifies by a name the top area thus created in the outline. Then, within **ter-**

FLOWCHART

Page 2:



Page 3:

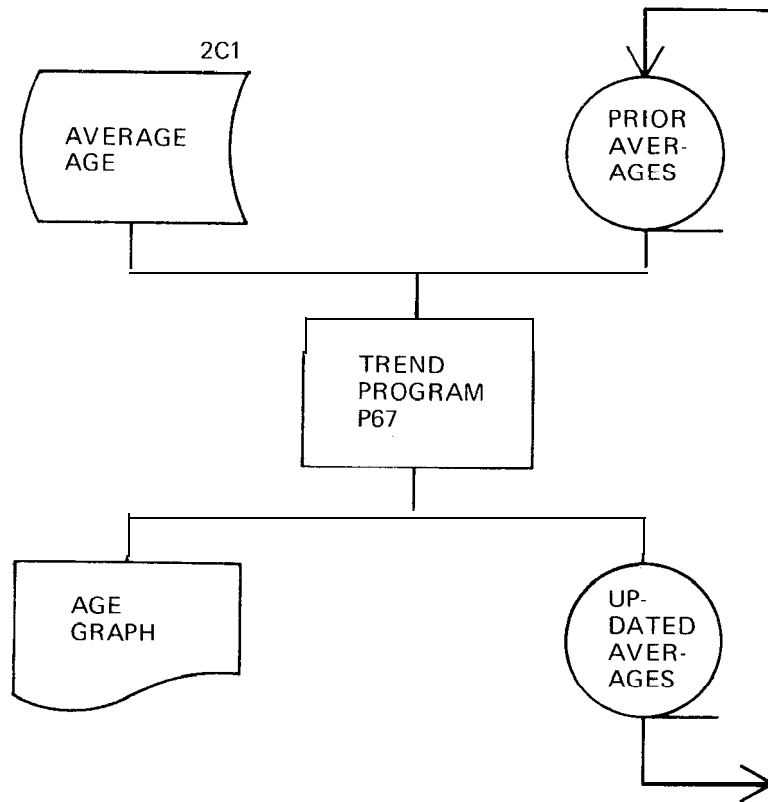


Fig. 9. Example of good practice in breaking a system chart. (From N. Chapin, *Flowcharts*, Petrocelli Books, New York, 1971.)

minimal connectors in flow diagrams or annotation outlines in system charts, he uses this name to identify the full sequence of operations he cited elsewhere in the flowchart. Adding location cross-references completes the substitution representation.

System Charts. The system chart variety of the flowchart identifies major sets or files of input and output data handled by the programs, people, and machines involved in a system. An example of a system chart is shown in Fig. 9. It is commonly prepared from the specialized outlines used to indicate the media or the equipment employed in the system to handle the data. Outlines for data that serve as inputs have flowlines drawn from them to a process outline. Outlines for data that are produced as outputs have flowlines drawn from the process

outline to them. An outline that represents output from one process may represent input to another process.

Each process identified in a system chart has at least one input and produces at least one output. Some processes, because of their complexity, require many inputs and produce many outputs. An output that serves as an input later for the same process is usually shown twice (once on each side of the process) with a connecting flowline (see Fig. 9). In general, the structure of a system chart is of alternating layers or process outlines and input/output outlines. An input outline starts the system chart; an output outline ends the system chart.

Flow Diagrams. The flow diagram variety of the flowchart describes algorithms, usually as they

are implemented in a computer program. Fig. 7 provides an example of a flow diagram for a simple statistical calculation. It is not necessary to identify the media or the equipment used for input or output because, under modern operating conditions, most of the input or output media and equipment typically can be changed from time to time without changing the program, and because computers can operate on data only within their internal storage units. Hence, the specialized media and equipment outlines find little use in flow diagrams. The basic input and output outline is normally the outline of choice.

Conventionally, a flow diagram begins with a terminal outline and ends with a terminal outline. Usually the first part of a flow diagram is concerned with initialization-operations to prepare the storage unit and the arithmetic and logic unit for the main operations.

The main operations and usually the middle part of a flow diagram involve reading in input data, performing some type of transformation action on the data, and producing some output data. Many decision or branching operations with divergent and later convergent flows are common.

The cleanup or final portion of most flow diagrams consists of completing the output data, terminating the use of the input and output equipment, and outputting any summary statistics. This structure is typically more complex than is the structure typical of the system chart, and is far less regular.

The specialized process outlines most used in flow diagrams are the decision, preparation, and predefined process outlines. The decision outline serves as the basis for selecting alternative flow paths, usually based upon comparison or test operations. The preparation outline is reserved for operations upon the program itself, such as setting the starting values of iteration controls, program switches, and the like. The role of the predefined process outline was noted previously.

In a flow diagram, cross-references are especially helpful to the user. This applies both to location cross-references within the flow diagram itself and to cross-references between the flow diagram and other descriptions or materials, such as the program listing.

Preparing and Using Flowcharts. Preparing flowcharts is usually a heuristic process of trying to set down a sequence of operations that might get the job done, given the data involved. Commonly, this is done in parts and in stages of

increasing precision. Often it is done by elaborating and correcting a former flowchart, and then using it as the basis for a new flowchart. The person preparing a flowchart normally continues this process until he has, given the level of detail he was striving for, completely described the program or system data transformation, step by step.

Most people draw flowcharts initially by hand, typically roughing them out in freehand form. When more neatly prepared flowcharts are desired, plastic templates are available to add a regularity and symmetry to the outlines, as shown in Fig. 10. Flowcharts produced as documentation after a program or system is completely implemented are typically prepared more neatly than are flowcharts produced as working documents- for use by programmers and analysts in implementing programs and systems. Detailed flow diagrams can be produced from source-coded programs, whether partial or complete, by the computer.

Using flowcharts is greatly aided by taking full advantage of the graphic character of the flow representation and of the shapes of the outlines. If the user knows what he is looking for, he can use the graphic features to scan even a lengthy flowchart very rapidly. Basic to such a scan, of course, is first identifying the type, either system chart or flow diagram. Sometimes location cross-references and annotation will have to be read to follow a flow. Once he has located the portion of interest, the user can turn his attention to the specific sequence shown, to the operations specified, and to the data acted upon, used in, or produced by the operations. For greater detail or for alternative descriptions, the user can follow the cross-references to related materials. In general, flowcharts are regarded as working documents upon which users freely enter comments, notations of difficulties encountered, changes made, and improvements possible in the program or system.

REFERENCES

1965. ANSI. "Graphic Symbols for Problem Definition and Analysis," *Communications of the ACM*, Vol. 8, No. 6, June, pp. 363-365.
1970. ANSI. *American National Standard Flowchart Symbols and Their Usage in Information Processing*, X3.5-1970. New York: American National Standards Institute.
1971. Chapin, Ned. *Flowcharts*. New York: Petrocelli Books.

N. CHAPIN

FORMAC. See SYMBOL MANIPULATION.

FORMAL LANGUAGES

For articles on related subjects see AUTOMATA THEORY; DECIDABILITY; LANGUAGE PROCESSORS; LANGUAGE TRANSLATION; and TURING MACHINE.
For article on related term see TREE.

Languages and Grammars. Formal languages are abstract mathematical objects used to model the syntax of programming languages or (less successfully) of natural languages such as English. For example, consider a simple English sentence, such as

THE MAN ATE THE APPLE.

Let us assume that individual English words are indecomposable objects. Then the study of English syntax attempts to answer the question: When is a string of words a grammatically correct English sentence? And when it is a sentence, how can it be parsed into its grammatical components?

To model this situation, we let V be a finite set of symbols, called a “vocabulary.” In the previous example, V contains the four indecomposable words (in this context, called “symbols” or “letters”): APPLE, ATE, MAN, THE. More generally, V might contain all English words and punctuation marks. Let V^* denote all finite-length strings of symbols from V . (It is mathematically convenient to

include in V^* the *empty* string of length zero.) Then a *formal* language L is simply a set of strings from V^* . For example, if V^* is the set of all finite sequences of English words, then L could be the subset of V^* consisting of all grammatically correct sentences. Although V is always finite, in most cases of interest L will be infinite, and we will wish to have a finitely specified way of generating, or recognizing, or parsing the strings in L .

The sample sentence given earlier can be parsed by the *treelike* diagram in Fig. 1, where (S), (NP), (VP), (A), (N), and (V) are six variables ranging over all *sentences*, *noun phrases*, *verb phrases*, *articles*, *nouns*, and *verbs*, respectively. Using the “rewriting”

<S> → <NP><VP>
<NP> → <A><N>
<VP> → <V><NP>
<A> → THE
<V> → ATE
<N> → MAN
<N> → APPLE

Fig. 2. Rewriting rules.

rules in Fig. 2, it is possible to generate our sample sentence from the variable (S). The generation proceeds as follows:

<S> ⇒ <NP><VP> ⇒ <A><N><VP>
⇒ <A><N><V><NP> ⇒ <A><N><V><A><N>
⇒ THE <N><V><A><N>
⇒ THE MAN <V> (A) (N)
⇒ THE MAN ATE (A) (N)
⇒ THE MAN ATE THE (N)
⇒ THE MAN ATE THE APPLE

With these rules we can also generate various improbable but grammatically correct sentences such as THE APPLE ATE THE MAN, and with more rules we could generate more sentences. Rewriting schemes of this sort were introduced by the linguist Noam Chomsky, who called them “context-free grammars.” Chomsky observed that these grammars are not good models for the syntax of natural languages, but it was soon discovered that they do closely *model* the syntax of programming languages, and for this reason they have been studied in great de tail.

To see a simple example of context-free rewriting rules that give rise to an infinite language, suppose that the vocabulary consists of two abstract symbols a and b , and let S be a variable. Then, using

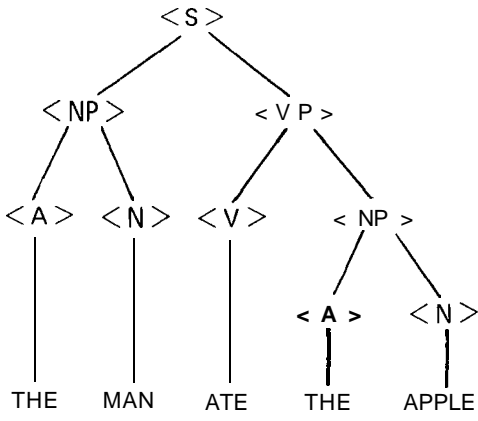


Fig. 1. Tree for parsing sentence.

the rules $S \rightarrow aSb$ and $S \rightarrow ab$, we can generate the infinite language

$$L = \{a^n b^n | n \geq 1\} = \{ab, aabb, aaabbb, \dots\}.$$

Rewriting rules of this type are called "context free" because they permit any occurrence of a variable within a string to be rewritten without regard to the context in which that variable occurs. By contrast, a rewriting rule like $aXab \rightarrow aYZcab$ is not context-free. It is called context "sensitive," since it allows X to be rewritten as YZc only when X occurs in the context $s_1 a _ abs_2$, where s_1 and s_2 are arbitrary strings.

To describe different kinds of grammars more precisely, let us define a *phrase-structure* grammar to be a quadruple $G = (V_N, V_T, P, S)$, where

1. V_N is a finite vocabulary of nonterminal symbols or variables.
2. V_T is a finite vocabulary of terminal symbols.
3. P is a finite set of rewriting rules (also called "productions") of the form $\alpha \rightarrow \beta$, where α is a **nonempty** string of variables and β is an arbitrary string of variables and terminal symbols.
4. S is a particular variable, called the "start" variable.

For all strings s_1 and s_2 , we may write $s_1 a s_2 \Rightarrow s_1 \beta s_2$ if $\alpha \rightarrow \beta$ is a production of the grammar G . Then the language generated by G is the set of all strings t of *terminal symbols* such that

$$S \Rightarrow s_1 \Rightarrow s_2 \Rightarrow \dots \Rightarrow s_n \Rightarrow t$$

for some choice of intermediate strings s_1, s_2, \dots, s_n . The intermediate strings may consist of both variables and terminal symbols.

Let α, α_1 , and α_2 denote arbitrary strings of variables and terminal symbols, and let A and B denote variables. If the productions in G have the specialized form $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, where β represents any **nonempty** string, then G is a *context-sensitive* grammar. (Frequently, a grammar is called context-sensitive if the productions merely have the form $\alpha \rightarrow \beta$, with β at least as long as α . These two definitions are in fact equivalent in the sense that the same collection of languages is generated.) If the productions in the grammar G have the form $A \rightarrow \alpha$, then G is context-free. If the productions have the form $A \rightarrow w_1 B$ or $A \rightarrow w_2$, where w_1 and w_2 are strings of terminal symbols, then G is right-linear. A language is called a "phrase-structure" language, or a "context-sensitive," "context-free," or "right-line-

ar" language, if it can be generated by a phrase-structure grammar, or a context-sensitive, context-free, or right-linear grammar, respectively.

To illustrate these ideas, let us examine the language $L = \{a^n b^n c^n | n \geq 1\}$. This language is not context-free because it cannot be generated by context-free productions. For example, suppose we attempt to generate L by imitating the way we used context-free productions to generate $\{a^n b^n | n \geq 1\}$. We might try the productions $S \rightarrow ABC$, $A \rightarrow aAB$, $A \rightarrow ab$, $B \rightarrow bBC$, $B \rightarrow bc$. But these productions generate

$$\{a^m b^m c^n | m \geq 1, n \geq 1\} = \{a^m b^{m+n} c^n | m \geq 1, n \geq 1\},$$

which is not the same as L .

The problem here is that the variables A and B interfere with each other. This lack of communication between variables always occurs in the intermediate strings produced by a context-free grammar because the context surrounding a variable has no influence on what that variable can generate. For this reason, no context-free grammar can generate the language L . Without some interaction between variables, there is no way to insure that the same number of a 's and b 's are produced, while simultaneously insuring that the same number of b 's and c 's are produced. However, the language L is context-sensitive because it can be generated by the context-sensitive productions

$$S \rightarrow ASBC, \\ S \rightarrow ABC, \quad CB \rightarrow DB, \quad DB \rightarrow DC, \quad DC \rightarrow BC, \\ AB \rightarrow AB', \quad B'B \rightarrow B'B', \quad B'C \rightarrow B'C', \quad C'C \rightarrow C'C', \\ A \rightarrow a, \quad B' \rightarrow b, \quad C' \rightarrow c.$$

A typical string in L , such as $aabbcc$, could be generated as follows: $S \Rightarrow ASBC \Rightarrow AABCBBC \Rightarrow AABDBBC \Rightarrow AABDCC \Rightarrow AABBBCC \Rightarrow AAB'BCC \Rightarrow AAB'B'CC \Rightarrow AAB'B'C'C \Rightarrow AAB'B'C'C' \Rightarrow aAB'B'C'C' \Rightarrow aabB'C'C' \Rightarrow aabbC'C' \Rightarrow aabbcc$. The diligent reader can verify that terminal strings which do not belong in L , like $aabcbc$, cannot be generated by these productions.

The four types of grammars (phrase-structure, context-sensitive, context-free, and right-linear) are also known as type 0, type 1, type 2, and type 3 grammars, respectively. They form a grammatical hierarchy, called the "Chomsky hierarchy." Among the four corresponding families of languages, the smallest family, the right-linear languages, is important because it turns out to consist precisely of those languages that can be recognized by finite-state automata. These languages arise in many different contexts, and they have the advantage of being very easy to parse.

FORMAL LANGUAGES

The next family in the hierarchy, the family of context-free languages, is important because context-free languages are good approximations to the syntax of programming languages, even though this syntax is usually a little too complicated to be completely captured by context-free grammars. Context-sensitive languages are powerful enough to encompass any complications in syntax that may have been missed by the context-free model, but they are so general that they are difficult to work with. As a result, they have been studied less than the other models, and various attempts have been made to add to the power of context-free grammars without resorting to the full strength of context-sensitive productions. These efforts have produced various kinds of grammars that are more powerful than context-free grammars, although they are unfortunately more complicated as well: programmed grammars, macro grammars, indexed grammars, and others.

The largest family of languages in the Chomsky hierarchy, the family of phrase-structure languages, is an important family because it represents the largest class with which one is likely to be concerned when modeling natural or artificial languages. This is so because the family of phrase-structure languages is in fact the same as the family of all recursively enumerable languages, i.e., of all languages L such that membership of a string w in L can be verified by some algorithm (or, more precisely, by some Turing machine).

Languages and Equations. We have noted that context-free languages are good approximations to the syntax of many programming languages. Consider the following very simple example of syntax specifications in *Backus-Naur form*, or *BNF*:

(digit) : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 (unsigned integer) : : = (digit) | (unsigned integer) (digit)

This means that (digit) and (unsigned integer) are the smallest sets of strings satisfying the following conditions: 0, 1, . . . , 9 are digits (i.e., they are in the set (digit)); any digit is an unsigned integer; and any unsigned integer followed by a digit is an unsigned integer. Rewriting these equations in a more algebraic form, we obtain:

$$D = "0" + "1" + \dots + "9"$$

$$U = D + U \cdot D$$

Consider these as abstract equations. What is their meaning? The unknowns U and D are variables whose values are languages; $X + Y$ denotes the union of the languages X and Y ; $X \cdot Y$ denotes the product of the languages X and Y , obtained by concatenating the strings in X with those in Y ; $X \cdot Y = \{xy | x \in X, y \in Y\}$; and "0," "1," etc., are constants denoting the languages consisting of just the single symbol 0, 1, etc. In general, the equations corresponding to BNF syntax descriptions can be more complicated than in our example. A typical equation might have the form

$$A = abBAAaAb + BaC + ba.$$

(The letters a and b are terminal symbols; A , B , and C are variables; and we have omitted the dot in products.) The operations $+$ and \cdot are roughly analogous to addition and multiplication of numbers; only \cdot is not commutative. (If X and Y are languages, $X \cdot Y$ is not generally the same as $Y \cdot X$.) If the product of languages were commutative, then we could write the term $abBAAaAb$ as $aabbA^3B$. This would be similar to a fourth-degree term in a polynomial expression, except that the variables range over languages rather than numbers and the coefficient $aabb$ is a string of symbols instead of a number. Since the product of languages is not commutative, we cannot rearrange terms in this way, but we can still regard these equations as polynomial equations in noncommuting variables. In general, the right-hand side of each equation will be a finite sum of terms, and each term will be a string of variables and terminal symbols. A set of such equations always has a unique smallest solution, so it always makes sense to speak of the "smallest sets of strings" U and D satisfying equations like those in our original example. The languages definable in this way by polynomial equations turn out to be precisely the context-free languages.

As a simple example, the language $\{a^n b^n | n > 1\}$ can be specified either as the language generated by the context-free productions $S \rightarrow aSb$ and $S \rightarrow ab$ or as the smallest solution of the equation $S = aSb + ab$. Incidentally, note that this equation is a first-degree or "linear" equation, since each summand contains at most one occurrence of a variable. Languages defined by such equations are called "linear" context-free languages. They can also be characterized as the languages generated by linear context-free grammars; i.e., by context-free grammars having productions of the form $A \rightarrow \alpha$, where the string α contains at most one occurrence of a

variable. It should now be clear why right-linear grammars are so named.

In view of the preceding discussion, any **programming** language whose syntax can be specified in BNF is context-free. Generally, most but not all of the syntax of a programming language can be specified in BNF. So languages such as Algol and Fortran are not quite context-free, but they are close to being so, and context-free languages are useful approximations to their syntax.

Languages and Automata. The four families of languages in the Chomsky hierarchy can be obtained from automata as well as from grammars. The phrase-structure languages are just the recursively enumerable languages, i.e., the languages accepted by Turing machines. (A Turing machine is a simple theoretical model of a computer. A Turing machine (T) accepts a string s if, when given s as input, T eventually reaches some designated accepting state. The language accepted by T is defined to be the set of all input strings that T accepts.) The context-sensitive languages can be characterized as those languages accepted by linear-bounded automata or *lba's*; the context-free languages are the languages accepted by pushdown automata; and the right-linear languages are the languages accepted by finite-state automata. For this reason, right-linear languages are sometimes called "finite-state" languages. Usually, however, right-linear languages are known as regular languages or regular sets. This terminology comes from Kleene's theorem, which states that a language is a finite-state language if and only if it can be represented by a regular expression. A regular expression is an expression that can be built up from individual strings by using the three operations $+$, \cdot , and $*$. The operations $+$ and \cdot are the operations of union and product introduced earlier. (The symbol \cup is sometimes used instead of $+$, and the \cdot may be omitted.) The operation $*$ is called the "Kleene closure" operation. If L is any set of strings, then L^* is defined to be the set of all strings that can be formed by concatenating together sequences of strings from L : $L^* = \{s_1 s_2 \dots s_n | n \geq 0, \text{ each } s_i \in L\}$. (By convention, the empty string is always in L^* .) For example, $(a + b)^* \cdot aaa \cdot (a + b)^*$ is a regular expression representing the set of all strings of a 's and b 's containing at least three consecutive a 's.

Let us consider the relation between context-free languages and pushdown automata a little more closely. A pushdown automaton is a **non-**deterministic device having a memory consisting

of a finite-state control and a pushdown stack. It receives its input one symbol at a time on request. Every context-free language L is the set of input strings accepted by some pushdown automaton P . In fact, we can always find a pushdown automaton P for L that operates in real time; i.e., one that uses up one input letter on every move. This means that P recognizes strings in L very quickly—in fact, in an amount of time proportional to the length of the input string. The catch is that P is a nondeterministic device. It is credited with accepting an input string w if there is *any* sequence of choices of moves (i. e., any sequence of "guesses") it can make while processing w that will lead it to an accepting mode, even though there may be other choices which do not lead to an accepting mode. But if we want to simulate P in the real world, we would systematically have to test every sequence of choices that P could make.

Since P might have several choices available to it on each move, this simulation could take exponentially more time than P does. This might suggest that the task of parsing a context-free language can be prohibitively time consuming, but in fact it is not. General-purpose, context-free, parsing algorithms can be designed to require only time n^3 , where n is the length of the input. One of the most popular such algorithms is Earley's algorithm. It takes time n^3 in the worst case, but for many context-free grammars it takes only a linear amount of time. The n^3 bound for an all-purpose, context-free parser can be improved slightly, but it is not yet known how much improvement is possible. In fact, it is conceivable (although very unlikely) that every context-free language can be parsed in linear time.

A nondeterministic pushdown automaton is a theoretical construct that is time consuming to simulate in the real world. So, in searching for classes of context-free languages that are easy to parse, it is reasonable to consider **deterministic** context-free languages—those languages that can be recognized by a deterministic pushdown automaton. As one might expect, all deterministic context-free languages can be parsed rapidly; in fact, in a linear amount of time. But not all context-free languages are deterministic. For example, the set of all binary strings (strings of 0's and 1's) which are palindromes is context-free but not deterministic because a pushdown automaton for this language must of necessity operate **some-**thing like this: Store the first half of the input string on the stack, guess when half the input has

been read, and use the stack to verify that the second half of the input agrees symbol by symbol, in reverse order, with the first half.

So, nondeterministic pushdown acceptors are more powerful than deterministic ones. Are the corresponding statements true for the other kinds of automata used to characterize the families of languages in the Chomsky hierarchy? For finite-state automata and for Turing machines, the answer is no. It is easy to show that the non-deterministic versions of these devices are no more powerful than the deterministic versions. In other words, the ability to make guesses may enable these devices to do their jobs more quickly, but it will not let them do anything that they could not have done without guessing. But for linear-bounded automata, it is still not known whether the nondeterministic version (which corresponds to the context-sensitive languages) is more powerful than the deterministic version. The answer is thought to be yes. That is, it is widely believed that deterministic *lba's* are not powerful enough to recognize all context-sensitive languages. But this question, called the "*lba* problem," has remained unresolved for more than a decade.

REFERENCES

1966. Ginsburg, S. *The Mathematical Theory of Context-Free Languages*. New York: McGraw-Hill.
1969. Hopcroft, J. E., and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.
1970. Gross, M., and A. Lentin. *Introduction to Formal Grammars*. New York: Springer-Verlag.
1972. Aho, A. V., and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*. Englewood Cliffs, N. J.: Prentice-Hall.
1972. Kain, R. Y. *Automata Theory: Machines and Languages*. New York: McGraw-Hill.
1973. Salomaa, A. *Formal Languages*. New York: Academic Press.

J. GOLDSTINE

FORTRAN. See PROCEDURE-ORIENTED LANGUAGES: Survey of.

FRONT-END PROCESSORS

For articles on related subjects see **CHANNEL**; **COMMUNICATIONS AND COMPUTERS**; **COMMUNICATION CONTROL UNIT**; **HOST SYSTEM**; **MULTIPLEXING**; and **PROCESSING MODES**.

For articles on related terms see **INTERUPT**; and **MODEM**.

A front-end processor is a small, limited capability, digital computer that is programmed to replace the hard-wired input functions of a central computing system (e.g., the control of remote terminals in a time-sharing system). The front-end processor thereby permits the host computer to perform its primary functions with little regard for the slower input/output activities associated with large-scale multiprogrammed or time-shared computing systems.

In addition to receiving and transmitting all data passing through a computing system, front-end processors also support a wide variety of functions, which might include :

1. **Data and/or format conversion**—the conversion of one or more incoming data codes and formats to that of the host system.
2. **Polling**—the determination by a front-end processor of a terminal's readiness to send or receive data.
3. **Assembly of characters and** messages—the assembly and disassembly of all data, input at varying line speeds and in synchronous or asynchronous formats, to insure that the host system receives only complete messages,
4. **Error control and editing**—the detection and possible correction of transmission errors as well as corrections initiated at the terminals, prior to reception by the host system.
5. **Fail-soft functions**—the ability of the front-end processor to keep parts of the system operating (such as terminals) when a major element of the system has failed.
6. **Queueing**—placing incoming messages in transmission order for processing by the host system, or in some cases queueing messages on auxiliary storage devices (spooling).
7. **Message switching**—a function of front-end processors that service more than one central processing unit (Fig. 1).
8. **Direct response**—the front-end processor may have the ability to respond to simple inquiries

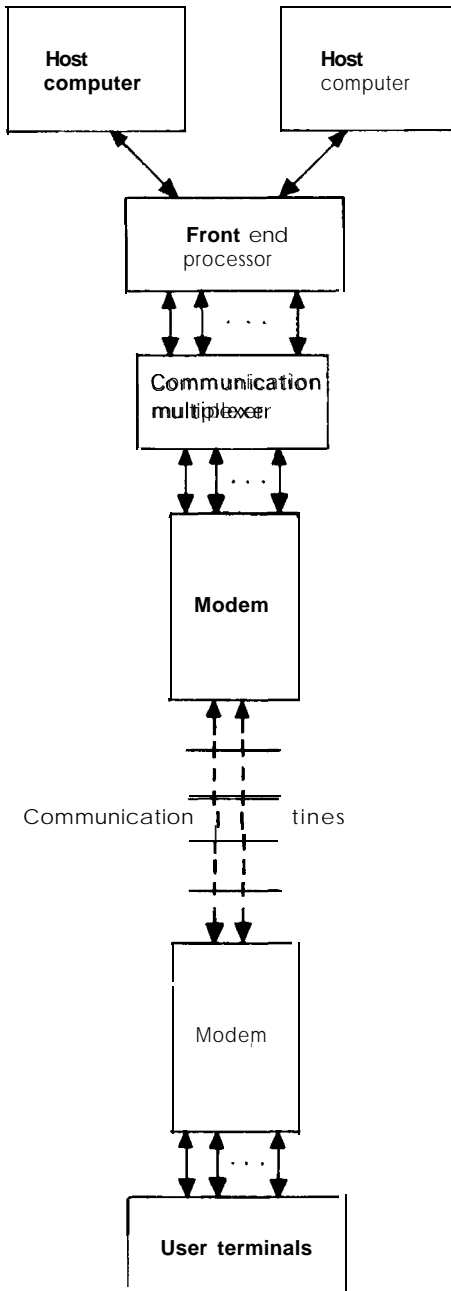


Fig. 1. A front-end message switcher.

directly without contact with the host system.

The basic components of a typical front-end processor are:

1. **Processor**—a stored program digital computer that has main memory, which may vary in size from

several hundred words to many thousand words, depending on the complexities of the specific application. Two important qualities required of a front-end processor are good facilities for bit manipulation and interrupt handling. The processor may or may not have its own on-line peripheral devices depending on the particular application.

2. **Central processor interface**—the hardware interface that allows the front-end processor to connect directly to the input/output channel of the host system. The host system is then able to communicate with the front-end processor as if it were a standard peripheral device controller.



Fig. 2. Honeywell Datanet 2000 processor.

3. **Communication multiplexer**—a device with programmable or hard-wired logic which produces logically independent data channels into the front-end processor's main memory from each transmission line serviced. The coordination of the data flow between the multiplexer and processor is handled by the front-end processor's interrupt system.

4. **Line interface units**—the hardware devices that link the communication multiplexer with the modems that terminate each of the communication lines.

FRONT-END PROCESSORS

5. *Software*—the programs that integrate the functions of the various hardware components of the front-end processor. Included in the software **package** are such functions as terminal, line and message control, system interface procedures, and whatever other functions are required by a particular **installation**.

The front-end processor can be a powerful and economical means of relieving a central processor of its time-consuming overhead activities by placing these activities under the control of an independent and parallel processing unit. Fig. 2 Shows such a unit, the Honeywell **Datanet 2000**.

A. I. **KARSHMER**

GAMES ON COMPUTERS

For articles on related subjects see ARTIFICIAL INTELLIGENCE; HEURISTICS; and SYMBOL MANIPULATION.

When the earliest digital computers were built, scientists immediately became fascinated with the possibility of having them play such games as chess, checkers, and tic-tat-toe. Although this sort of activity proved to be a great deal of fun, the scientists were not just playing around; as it turns out, there are several good reasons to study game playing by computers.

The first reason relates to the popular conception of computers as "giant brains." Even the earliest digital computers could do arithmetic and make decisions at a rate thousands of times faster than humans could. Thus, it was felt that computers could be set up to perform intelligent activities such as to translate French to English, recognize sloppy handwriting, and play chess. At the same time, it was realized that if computers could not perform these tasks, then they could not be considered intelligent by human standards. A new scientific discipline arose from these considerations and became known as "artificial intelligence."

A second reason involves man's understanding of his own intelligence. It is conjectured that computer mechanisms for game playing will bear a resemblance to human thought processes. If this is

true, then game-playing computers can help us understand how human minds work.

Another reason for studying games is that they are well-defined activities. Most games use very simple equipment and have a simple set of rules that must be followed. Usually, the ultimate goal (winning) can be very simply defined. Thus, a computer can be easily set up to know the rules of any board game or card game. This allows the computer scientist to devote more effort to the problem of getting the computer to play an intelligent game.

There is also a practical payoff from computer game-playing studies. Specific techniques developed in programming a computer to play games have been applied to other more practical problems. To cite a few, methods of search, which are used to consider alternative moves in chess, have been adapted to find the correct path through a switching network or the correct sequence of steps for an assembly line. Learning methods developed for a checker-playing program have been used to recognize elementary parts of spoken speech. It is felt that the mechanisms of intelligence are general purpose, and therefore the borrowing of techniques from one application to another will continue in the field of artificial intelligence.

Basic Techniques. The fundamental reason for the ability of computers to play a variety of games is that computers have the ability to represent arbitrary situations and processes through the use of symbols and logic operations. For example, one can

GAMES ON COMPUTERS

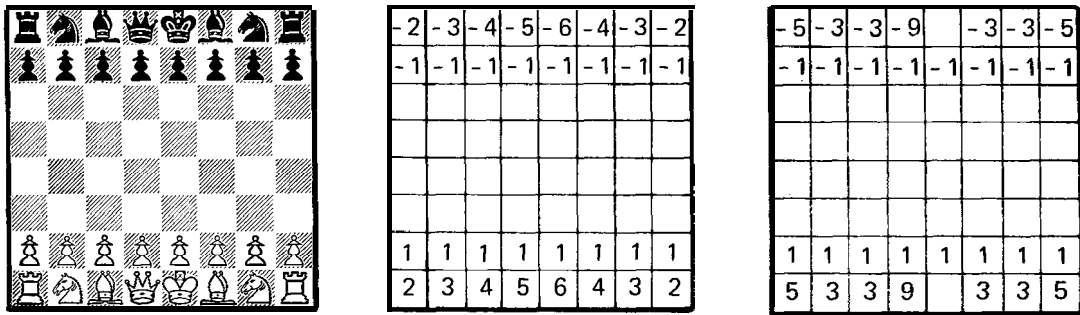


Fig. 1. Computer representation of a chess position. In the second array, numbers are used to represent the various pieces. The third array represents the values of the pieces for use by the computer in evaluating trades.

set up a chess position inside a computer by means of an 8 by 8 array of integers, and tentative moves can be made by computer instructions that change the positions of the numbers in the array (Fig. 1). This capability is extremely general. That is, the symbols could represent checker pieces, or with a slight rearrangement, they could be playing cards for poker or bridge.

Fig. 1 also shows the representation of derived information. The values of the pieces are stored in another 8 by 8 array for use by the computer. In effect, they are part of the computer's "knowledge" of the values of chess pieces. (The king may be considered to have an infinitely large value.)

Since symbols can be used to represent the objects of a particular game, computer instructions can be written by a programmer to specify the procedures for playing the game according to the rules and also for playing the game according to a strategy. In order for a set of procedures to be programmable, it is usually sufficient that they be defined in enough detail so that they can be translated into a computer language such as **Fortran**. For the purposes of this exposition, some game playing algorithms will be stated using English words in place of computer language. The game of tic-tat-toe, for example, can be played perfectly by the following algorithm, in which the word "row" refers to a row, column, or diagonal.

ALGORITHM A (THE COMPUTER PLAYS X)

A1. Perform the first applicable step which follows*

A2. Search for two X's in a row. If found, then make three X's in a row.

A3. Search for two O's in a row. If found, then block them with an X.

A4. Search for two rows that intersect with an empty square, each of which contains one X and no O's. If found, then place an X on the intersection.

A5. Search for two rows that intersect at an empty square, each of which contains one O and no X's. If found, then place an X on the intersection.

A6. Search for a vacant corner square. If found, then place an X on the vacancy.

A7. Search for a vacant square. If found, then place an X on the vacancy.

The algorithm is perfect in the sense that it will find a forced win if it exists and it will never lose. This algorithm may be called a rejection scheme because the first applicable step (following A1) is to be performed and all other steps are rejected or ignored (Fig. 2). A computer can be easily programmed to execute such an algorithm.

For another example, consider the following game, a special case of the game of Nim. It is played

X	O	X
	O	2
O	1	X

Fig. 2. In the position shown here, algorithm A would choose a move at square 2 for a win rather than at square 1, to block the opponent win. This is done because step A2 precedes step A3.

with 13 matches; two players remove matches in turn until one player is forced to take the last match. A player may remove only one to three matches in a single turn, and the player who removes the last match is the loser. This is an algorithm for perfect play.

ALGORITHM B (THE COMPUTER PLAYS SECOND)

- B1. Let n be the number of matches taken by the opponent,
- B2. Remove $(4 - n)$ matches.
- B3. If the game is not over, go to Step B1.

Both tic-tat-toe and Nim are simple examples of a large number of games classed as two-person games of skill. An essential feature of these games is that both players have perfect information about the current state of the game. Chess, checkers, and GO are well-known games of pure skill. It can be shown mathematically that there is an optimal strategy for each player and that its application always gives the

same result. In the case of tic-tat-toe, the result is a draw. In the case of the match game, the second player always wins.

In order to show that an optimal strategy exists for two-person games of skill, the principle of *minimax* must be explained. If the state of a game is represented by a circle and the moves from that state are represented by lines (Fig. 3), then a tree can be obtained which represents the set of all possible games. The end nodes of this tree (Fig. 4) can all be labeled with the terms "win," "loss," or "draw" for the first player. Now consider any node that is followed only by labeled nodes. If that node corresponds to the first player's move and it is connected to a node labeled "win," then it may be labeled with the term "win." It may be labeled with a "draw," if it is connected to a draw. Otherwise, it is labeled with a "loss." If it is the second player's move from a position, then a loss is most preferred. This procedure can be repeated to back up the values W, L, and D to the top of the lookahead tree.

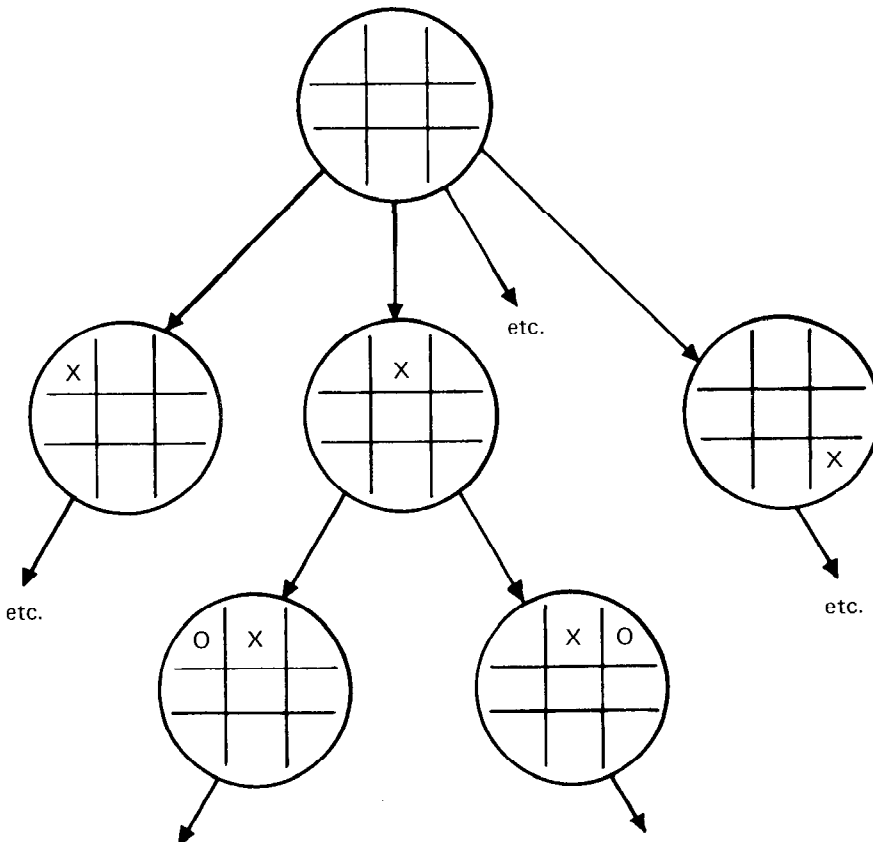


Fig. 3. Part of the lookahead tree for tic-tat-toe. Circles represent game positions and arrows represent moves by X or O.

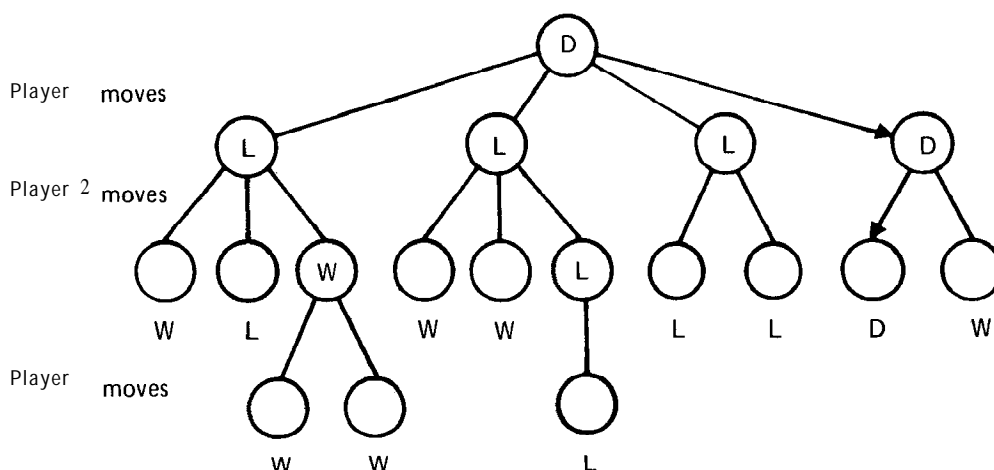


Fig. 4. Illustration of the minimax procedure. The values at the bottom are calculated by an evaluation function. The backed-up values shown in circles reflect the result of optimal play. The arrows show the path of optimal play.

Optimal strategy consists of following the path taken by the letter W, L, or D, which is backed up to the top. In other words, a player makes the best move, based on the assumption that his opponent will make the best reply, and his opponent's reply assumes that the player will make the best counter-reply, etc. (The best outcome is guaranteed, of course, even if the opponent makes less than optimum moves.)

Thus, since all possible chess games can be expressed in a tree like that in Fig. 3, it is known that there is a perfect strategy for chess, which guarantees a win or a draw for one player. Of course the strategy has never been found. It is the combinatorics of game playing which prevent the discovery of perfect strategies. In chess, each player, when it is his turn to move, has on the average 30 legal moves resulting in 30 different positions. If the opponent also has 30 replies to each of those moves, then 900 positions result. This sort of calculation gives an estimate of 10^{125} as the size of the lookahead tree for chess (the number of paths from the top of the tree to the terminal positions). If a computer could examine a billion positions per second, it would still take 10^{110} years to examine the entire lookahead tree to determine the optimal strategy.

A second class of game involves no skill at all, and a player's success depends only on luck. Examples are craps and roulette, in which the roll of dice or drop of a ball determines whether a bet is won or lost. A third and most important class of game involves a mixture of skill and luck of varying degrees. This includes games such as poker, bridge,

backgammon, and monopoly, which are affected by the distribution of cards or the roll of dice, although these randomizing features are generally overcome in the long run by the skill of a player. Computers can be set up to play these games of chance, but it is usually more difficult to represent the state of the game when unknown or probabilistic factors are present.

Chess-Playing Programs. The means by which a computer could be set up to play chess were first described by Claude Shannon (1950), a mathematician and engineer. He proposed that a minimax search be performed to find a good move, but because a complete minimax search was not possible, he proposed that the search terminate at a fixed depth of about four plies (two moves for each side). If there is a choice of 30 moves from each position, then only 810,000 positions have to be examined for a depth-four search.

The key part of Shannon's proposal was the use of a programmed evaluation function that assigned a numerical measure of the "goodness" of the various board positions at depth four. Thus, instead of using three values (win, draw, loss), minimax would operate on a whole range of values with win and loss as the extreme end points. A simple example of such an evaluation function would be piece advantage. This function would simply sum all numbers in the third array of Fig. 1. A more comprehensive evaluation function would have to assess positional considerations such as pawn structure, center control, mobil-

ity, interaction of pieces, etc. Shannon proposed that the evaluation function be a linear sum of the terms expressing the assessment in a quantitative form.

The resulting change to **minimax** is easy to implement. When the computer has a choice of moves, it always chooses the position with maximum value. The opponent always chooses the position with a minimum value. Thus, the values placed at depth four in the lookahead by the evaluation function can be backed up to level 3 and then to level 2, which determines the best move.

To complete his proposal, Shannon suggested that the computer should not examine as many as 30 moves from a given position. It is usually the case that a majority of moves are ridiculous; if the computer can be given some criteria for selecting or rejecting moves, then the lookahead tree will be "pruned" to a much smaller size. For example, a depth-four lookahead that considers 15 moves from each position has to examine only 50,625 positions.

Unfortunately, the computers of 1950 were too crude for the implementation of Shannon's ideas. In 1953, A. M. Turing tried to simulate the operation of a chess computer by hand. The result was poor play by any standard (and the hand simulation resulted in a logic error as well). The first actual computer program was reported by a group of scientists at Los Alamos (Kister et al.). Their program was set up to play on a 6 by 6 board (omitting bishops) and did not allow castling or *en passant* capture. The program did a **minimax** search to depth four, but did not prune out bad moves. It was able to beat a human player who had 20 games previous experience.

The first reasonable computer chess player was described by Bernstein and Roberts in 1958. The Bernstein program examined seven moves from each position to a fixed depth of four, resulting in the examination of 2,401 positions in about 2 minutes of computer time. The selection of seven moves for consideration was made according to the following criteria:

- C1. If in check, capture the checking piece, interpose a piece, or move the king away.
- C2. If exchanges that gain material are possible, make the capturing move to start the exchange.
- C3. If castling is possible, castle.
- C4. Develop a knight or bishop.
- C5. Occupy an open file with a queen or rook.
- C6. Move a piece into a pawn chain so that it cannot be attacked by a pawn.

- C7. Move a pawn.
- C8. Move a piece.

Regardless of the number of possible moves from a given position, these criteria are used to select seven moves for further consideration. The evaluation of board positions at depth four was a sum of four terms: mobility, material, area control, and king defense.

Newell, Shaw, and Simon also described a chess program in 1958 which played a passable amateur game according to the complete rules of chess. Like the Bernstein program, this one selected reasonable moves to limit the size of the lookahead search, but the move selection was made according to "goal generators," which attempted to direct the computer's play toward one or another specific goal.

By 1970, several programs appeared which raised computer chess players to the level of grade C tournament play. The increased proficiency of these programs is due mainly to faster computers and greater programming effort rather than to conceptual breakthroughs in the art of chess programming.

One new idea was described by Zobrist (1973) and Carlson. Apparently, the Shannon-type chess programs are limited by the amount of chess knowledge that can be programmed into a computer, using standard computer languages. Zobrist and Carlson designed a special language for the description of chess knowledge and implemented a program that can interpret that language to play a better game of chess. Figure 5 illustrates the use of their chess language. Their approach is called "advice taking," since it allows a chess player to give advice to the program by means of the chess language.

A computer chess tournament is held each year in conjunction with the ACM annual conference (the Association for Computing Machinery is the main professional organization for computer scientists). Since the participating computers are scattered across the country, moves are communicated via teletype terminals and telephone. This tournament has been held yearly from 1970 through 1975, and has been won every year except 1974 by a series of programs, Chess 3.0 to Chess 4.4, written by David Slate, Larry Atkin, and Keith Gorlen from Northwestern University. The display at the Northwestern computer terminal is shown in Fig. 6. In 1974 the tournament was won by the program **TREEFROG**, which was written by a group at the University of Waterloo. The first international computer chess tournament, held at the IFIP Congress in Stockholm in the summer of 1974, was won by a Russian program, Kaissa.

GENERAL REGISTER

A common variation of the above basic scheme is compacting garbage collection. Here, the second stage includes physical rearrangement of data cells so that all the garbage is compressed into a contiguous array. This process requires an extra pass through memory to correct existing pointers to data that have been moved. This extra work is sometimes worthwhile because an available space **array** can be utilized more efficiently than an available space list, especially in paged and swapping memory systems.

B. RAPHAEL

GENERAL REGISTER

For articles on related subjects see **ARITHMETIC-LOGIC UNIT**; and **REGISTER**.

For article on related term see **CHANNEL**.

A general register is a storage device that holds the input (operands) and the output (results) of the various functional units of a computing system. It is also used for temporary storage of intermediate results.

The width of the register is directly related to the width of the operational units, as it appears to the programmer, and does not necessarily reflect the width of the main-memory addressable unit. Thus, in the IBM/370, for example, the general registers are 32 bits wide, although the memory is addressed in 8-bit units.

The functional units, referred to in the definition, usually include the arithmetic unit, the memory, the control unit, and various input/output processors.

The registers operate at a speed that is directly connected to the speed of the units they serve. Their speed must be such that they do not slow down in any considerable way the functional units connected to them. In this sense they are the highest-speed storage part in the **hierarchy** of stores present in a computer.

Among a multitude of hardware reasons for the presence of general registers, one should note in particular their role in reducing the average number of bits needed to specify the operands in a computer program.

As their name implies, the usage of general registers is varied. They may serve as **arithmetic** registers, in which case they function as dedicated parts of the arithmetic unit. If we denote registers by

R , then a typical arithmetic instruction will be $R_i \leftarrow R_j \text{ O } R_k$, where o stands for any arithmetic operation, and i, j, k may be either distinct or equal (e.g., $R_2 \leftarrow R_2 + R_3$).

The general registers may also serve as: shift registers; index registers, in which case they serve as input to the memory unit; input/output registers, in which case they hold parameters that specify channels; or channel command registers, etc.

The number of general registers varies widely between 1 to 256 (at the time of this writing). The numbers represent today's architecture and hardware trade-offs, and are not to be taken as magic numbers. There are also computers that possess more than one set of general-purpose registers (and the programmer may switch between them), and computers that possess no general registers at all.

G. FRIEDER

GENERATIONS, COMPUTER

For articles on related subjects see **DIGITAL COMPUTERS**: Early, and Contemporary and Future; and **MANUFACTURERS, COMPUTER**.

For articles on related terms see **CONTROL DATA CORPORATION 6000 SERIES**; **IBM 360-370 SERIES**; **IBM 1400 SERIES**; and **INTEGRATED CIRCUITRY**.

In discussions of the history of electronic computers, it is convenient to refer to at least three computer generations.

The first generation is characterized by the use of vacuum tubes as active elements. This generation started with one-of-a-kind computers in university and government research laboratories. **Mercury**-delay lines and electrostatic storage tubes were the typical memory devices in the early systems.

The development of a reliable magnetic core memory was a major turning point in the first generation. The IBM 704 is an impressive example of the advanced hardware and software technology of that period. The latter part of the first generation also saw the introduction of many computers that used magnetic drums as their main storage.

The second generation is characterized by the use of transistors as active elements. The first important transistorized computers were delivered in 1959, and vacuum tubes rapidly disappeared from computer systems. The second generation was char-

GLOBAL AND LOCAL VARIABLES

acterized by some powerful computers: **Larc**, **Stretch**, **IBM 7090**, **Philco 2000**, **CDC 3600**, etc., and many small systems such as the **IBM 140 1**, **RCA 301**, **CDC 160A**, etc.

The distinction between the second and third generation is not nearly as clear-cut as that between the first and second. Computers that use integrated circuit technology are, by definition, third-generation computers, but some of the most powerful computers of the third generation use discrete component technology. It is capability and performance rather than circuitry that makes a large computer a member of the third generation. They are characterized by their ability to support multiprogramming and multiprocessors with a rather elaborate disk-based operating system. A typical third-generation operating system on a large computer handles multiple local and remote job streams and can support a variety of remote on-line terminals.

The third generation is generally considered to have started in 1964. Improvements in technology since that time, especially a trend toward the use of large-scale integration, could be considered the beginning of a fourth computer generation. Most authorities, however, consider the computers introduced since 1969 to be "late third generation" computers. They look for a more significant breakthrough, such as an electronic peripheral storage system to replace disk storage, to characterize a fourth generation.

Many different models of third-generation computers exist at the present time. Some of the better known ones are the **IBM 360** and **370** series, **UNIVAC 1108** and **1110**, **Honeywell 6000** series, **Control Data 6000**, **7000**, and **Cyber 70** and **170** series, **Burroughs 5700** and **6700**, **Digital Equipment Corporation's PDP-10** and **PDP-11**, and minicomputers (as well as some larger computers) manufactured by **Data General**, **Hewlett Packard**, **Varian**, **Texas Instruments**, **Microdata**, and many others.

S. ROSEN

GIGO

GIGO (garbage in-garbage out) is a popular acronym in computing. A more precise statement of the principle involved is "output is a function of the input and the instructions." The implication is that if the input data is erroneous, or the sequence of instructions is illogical, or both, then it should not be

astonishing that the results make little sense. Only if all parts of the computing activity are precisely correct can one expect useful results.

F. GRUENBERGER

GLITCH

For article on related subject see **BUG**.

The term "glitch" is a small error of any kind—a bug.

F. GRUENBERGER

GLOBAL AND LOCAL VARIABLES

For articles on related subjects see **BLOCK STRUCTURE**; and **PROCEDURE-ORIENTED LANGUAGES**, Programming in.

The quantity (or quantities) referred to by a given variable name in a computer program can generally be accessed (i.e., used or changed) only in certain parts of the program. The domain of the program during which a variable name can be accessed is called the "scope" of the variable.

In a block-structured language, the scope of a variable is the block in which it is declared, but excludes any subblocks that are internal to the defining block and in which the same variable name is declared. This is illustrated in Fig. 1, which shows the schematic of an Algol program with an outer block **L1** and an inner subblock **L2**, which in turn contains two further subblocks **L3** and **L4**. Also shown in Fig. 1 is the scope of each variable. Note in particular that a variable like **C**, defined in the outer block, has a scope **L1** minus **L4** because **C** is defined again in **L4**.

A variable in a block in which it is defined, like **G** in block **L4** in the example, is said to be *local* to that block, and is therefore a local variable. Correspondingly, variable **A** is *global* to block **L4**, since it is defined outside this block, although it may be referred to in the block. The variable **C** defined in the outer block is also global to block **L4**, but it cannot be referred to in **L4** because of the declaration of **C** in block **L4**, the latter (but different) **C** being local to **L4**.

GRAMMAR

In a language such as Fortran, where sub-programs are separate from the main program, a local variable in a subprogram is one that is defined and used only in the subprogram, while a global variable is one used to communicate with the main program as the name of an input argument from the main program, an output argument to the main program, or both.

```
L1: begin
  real A, C, D; real array B[ 1:10];
  L2: begin
    real D, E; real array F[ -4:6,1:12];
    L3: begin
      real F, G;
      .
      .
    end L3;
    L4: begin
      real B, C, G;
      .
      .
    end L4;
  end L2;
end L1;
```

Variable Name	Label of Defining Block	Scope of Name
A	L1	L1
B	L1	L1-L4
C	L1	L1-L4
D	L1	L1-L2
D	L2	L2
E	L2	L2
F	L2	L2-L3
F	L3	L3
G	L3	L3
B	L4	L4
C	L4	L4
G	L4	L4

Note: L1-L4, for example, means those parts of the program in block L1 but *not* block L4.

Fig. 1. Scope of variable names.

REFERENCE

1971. Ralston, A. *An Introduction to Programming and Computer Science*. New York: McGraw-Hill.

J. A. N. LEE AND A. RALSTON

GRAMMAR. See GRAMMAR, GENERATIVE; GRAMMAR, REDUCTIVE; and LANGUAGE PROCESSORS.

GRAMMAR, GENERATIVE

For articles on related subjects see GRAMMAR, REDUCTIVE; PARSING; and PROGRAMMING LINGUISTICS.
For article on related term see META VARIABLE

A grammar is a set of rules that describes the valid forms of a language on the basis of a set of the “parts of speech” (formally called the set of “meta-variables,” or “phrase names”) and the alphabet or character set of the language. Grammars are most commonly classified into two groups: *context sensitive* and *context free*. In the case of context-sensitive grammars, the rules are applicable only when a metavariable occurs in a specified context—for example, the modification of verbs to their plural form in the context of plurality in the rest of the sentence in natural languages. By contrast, in a context-free grammar any occurrence of a metavariable may be replaced by one of its alternatives, irrespective of the other elements in the language. Most programming languages appear to be describable in context-free grammars except where certain declaratives are required, such as the declaration of array dimensions or the specification of a procedure to support a procedure reference. In the discussion that follows, we will restrict ourselves to context-free grammars.

Given a starting phrase name, such as *sentence*, a generative grammar specifies a sequence of replacements that can be applied to that name to form an instance (in this case a *sentence*) in the language. For example, consider the following small grammar:

sentence = *noun-phrase verb-phrase*
noun-phrase = *article noun*
verb-phrase = *verb noun-phrase*

and

article = the, a
noun = cat, milk
verb = drank

where the italicized elements are metavariables

and the nonitalicized elements are from the alphabet of the language. Using these rules, the sentence

The cat drank the milk.

can be generated by the following sequence:

sentence → *noun-phrase verb-phrase*.
 → *article noun verb-phrase*.
 → the *noun verb-phrase*.
 → the cat *verb-phrase*.
 → the cat *verb noun-phrase*.
 → the cat drank *noun-phrase*.
 → the cat drank *article noun*.
 → the cat drank the *noun*.
 → the cat drank the milk.

Equally, the sentences "the milk drank the cat" and "the cat drank the cat" can be generated, since they have the required underlying syntactic (grammatical) structure.

Similarly, consider the following grammar for simple forms of arithmetic expressions in higher-level languages (where the vertical bar is to be read "or"):

add-q = + *I*
mult-op = *//
exp-up = **
primary = *constant* | *variable*
factor = *primary* | *primary exp-up primary*
term = *factor* | *factor mult-op factor*
arithmetic-expression = *term add-op term*

and where constants and variables then have usual definitions in computer languages. Then the expression

$A + B * C ** D$

could be generated as follows:

arithmetic-expression → *term add-op term*
 → *factor add-up term*
 → *primary add-op term*
 → *variable add-op term*
 → *A add-op term*
 → *A + term*
 → *A + factor mult op-factor*
 → *A + primary mult-op factor*
 → *A + variable mult-op factor*
 → *A + B mult-op factor*
 + *A + B * factor*

→ $A + B * \textit{primary exp-op primary}$
 → $A + B * \textit{variable exp-op primary}$
 → $A + B * C \textit{exp-op primary}$
 → $A + B * C ** \textit{primary}$
 + $A + B * C ** \textit{variable}$
 → $A + B * C ** D$

REFERENCES

1963. Chomsky, N. "Formal Properties of Grammars," in *Handbook of Mat. Psych.*, Vol. 2. New York: John Wiley, pp. 323418.
 1969. Hopcroft, J. E., and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Reading, Mass.: Addison-Wesley.

J. A. N. LEE

GRAMMAR, REDUCTIVE

For articles on related subjects see **GRAMMAR, GENERATIVE**; **PARSING**; and **SYNTAX, SEMANTICS AND PRAGMATICS**.

For article on related term see **META VARIABLE**.

A reductive grammar is a set of syntactic rules for the analysis of strings to determine whether the strings exist in a language. As contrasted with a generative grammar, a reductive grammar is designed to permit the reduction of the string to a recognizable metavariable.

In the process of syntactic analysis, a reductive grammar permits the analyzer to commence its operations at the level of the string and to work toward the target of the root symbol in the language. For example, a reductive grammar would take the string

$A + B * C ** D$

and with the reverse series of transformations used with a generative grammar, would achieve the target of *arithmetic-expression*. Such an analysis is known as a "bottom-up" analysis, in contrast to the "top-down" analysis of a generative grammar, which starts from the root symbol (e.g., *arithmetic-expression*) and works down toward the string being analyzed.

In practice, computer language processors normally use a combination of the reductive and

GRAPH THEORY

generative techniques when doing the syntactic analysis of strings.

J. A. N. LEE

GRAPH THEORY

For articles on related subjects see **ALGORITHMS**, **ANALYSIS OF**; **COMPUTATIONAL COMPLEXITY**; **DATA STRUCTURES**; and **TREE**.

For article on related term see **HEURISTICS**.

Informally a graph is a collection of points, any pair of which may or may not be joined by a line. Graph theory is the study of these objects. A graph may be represented by a diagram such as that in Fig. 1.

The uses of graph theory in computer science are diverse, and they include applications such as scheduling in operating systems and elsewhere, resource allocation, flowchart representation, information retrieval, and even sorting. The algorithms developed to solve combinatorial and graph theoretical problems have recently been found to be also of theoretical interest in the area of computational complexity. We attempt here simply to give basic definitions and to summarize the current state-of-the-art in the algorithmic area.

The terminology of graph theory varies from author to author and from application to application; the reader should take care to know what concept is intended when reading articles from diverse fields. We attempt here to use what has become more or less standard vocabulary, and to mention alternative words that are common.

It is frequently pointed out that graph theory had its beginnings with Euler and the Konigsberg bridge problem. The problem is to find a way of taking a walk (in Konigsberg) and to cross each of its seven bridges (Fig. 2) exactly once. Euler showed such a walk to be impossible, by noting that each of the land masses *A, C, D* has an odd number (three) of bridges. It is noteworthy that Euler's solution to the generalization of this problem (made in 1736), which is known as finding an *Eulerian path* when one exists, remains the basis of the best algorithm known today for the solution to this problem.

A graph $G(V, X)$ is a finite, **nonempty** collection V of points (nodes, vertices) and a prescribed set X of unordered pairs of distinct points. Such a pair $\{v_1, v_2\}$ is called an "edge" (branch, arc, line). The diagram of a graph is sometimes referred to as the graph itself.

A graph is called "labeled" (see the rightmost example in Fig. 1) if its points are distinguished by names. A *directed graph*, or *digraph* (Fig. 3), is a finite **nonempty** set of points V and a set X of ordered pairs of distinct points for edges. A digraph is said to be *oriented* if both (v_i, v_j) and (v_j, v_i) do not occur in X .

Note that the preceding definitions do not allow an infinite set of points, multiple edges, or loops (a line from a point to itself). Some authors allow all or some of these in graphs and reserve some special word (linear graph, for example) to mean the structure we define as simply a graph.

There are two common ways to represent a graph or digraph in a computer: (1) by its *adjacency matrix*, and (2) by its *adjacency structure*. The adjacency matrix for a graph on n vertices is $A = (a_{ij})$, $i, j = 1, 2, \dots, n$, where $a_{ij} = 1$ if vertex v_i is *adjacent* to v_j [i.e., if (v_i, v_j) is an edge of the graph], and zero otherwise. The adjacency structure is the listing for each vertex of all other vertices adjacent to it. The adjacency structure has proved to produce faster

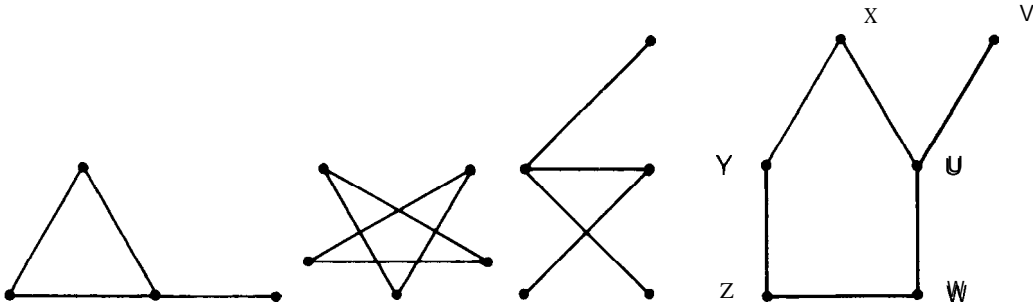


Fig. 1

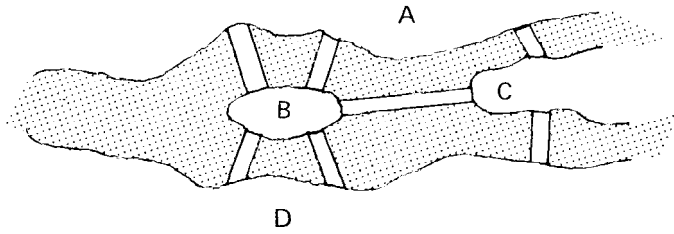


Fig. 2

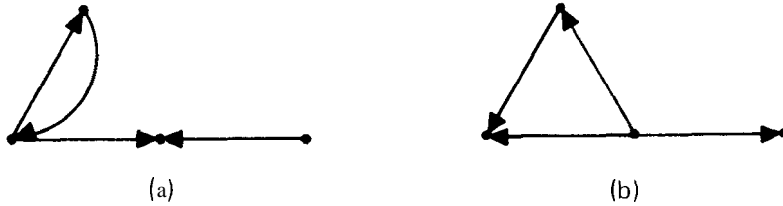


Fig. 3. (a) A digraph. (b) An oriented graph.

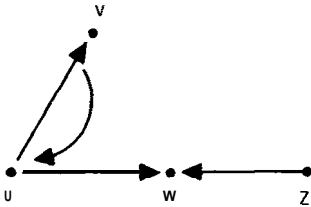


Fig. 4

algorithms on a random access, nonparallel machine.

For the labeled digraph in Fig. 4, the adjacency matrix is

$$\begin{matrix} & \begin{matrix} u & v & w & z \end{matrix} \\ \begin{matrix} u \\ v \\ w \\ z \end{matrix} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{matrix}$$

and the adjacency structure is

$u: v, w$
 $v: u$
 $w: \{\text{empty}\}$
 $z: w$

Similarly, for the labeled graph in Fig. 5, we have the adjacency matrix:

$$\begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{matrix}$$

and adjacency structure

1: 2,4
 2: 1,3
 3: 2,4
 4: 3,4

The reader will note that if no orientation is given to the edges [i.e., " (v_i, v_j) is an edge" means that v_i is adjacent to v_j , and conversely], then the adjacency matrix is symmetric about the diagonal, which is zero when no loops are present. Also, any permutation on the rows and columns (PAP,

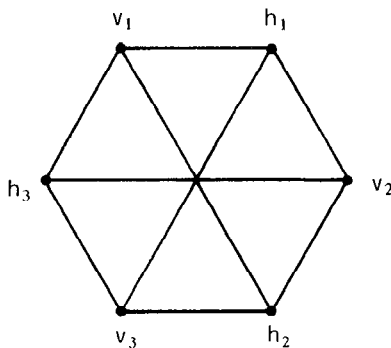
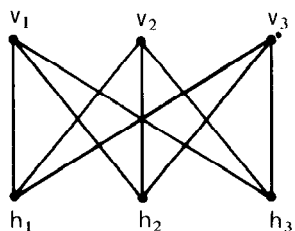


Fig. 6. (a) $K_{3,3}$, (b) K_5 .

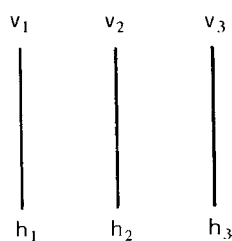
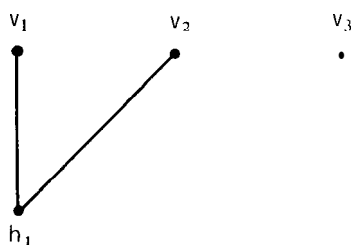


Fig. 7. (a) A subgraph of $K_{3,3}$, (b) A spanning subgraph of $K_{3,3}$.

where P is a permutation matrix) simply represents the same graph with a different labeling.

A *subgraph* of a graph G is a graph whose points and edges are all in G ; a *spanning subgraph* is a subgraph that contains all the points of G . Two graphs, G_1 and G_2 , are *isomorphic* if there exists a 1-1 correspondence between their point sets which preserves adjacency. The diagrams of isomorphic graphs may appear quite different (see Fig. 6).

The graph $K_{3,3}$ and the related graph K_5 of Fig. 6 are famous because the mathematician Kuratowski showed that a graph is *planar* (roughly can be drawn in the plane without any lines crossing) if and only if it contains no subgraph homeomorphic to $K_{3,3}$ or K_5 . This theorem has not proved useful in computer algorithms to determine whether or not a graph is planar,

For examples of a subgraph, and a spanning subgraph of $K_{3,3}$ see Fig. 7(a) and (b).

A set of vertices $\{v_0, v_1, \dots, v_n\}$ in a graph G is a *walk* of length n if $\{v_i, v_{i+1}\} i = 0, 1, \dots, n-1$ is an edge of G and a *path* if all v_i are distinct. Some writers use the word "path" when a given vertex is allowed to appear more than once in the set $\{v_0, v_1, \dots, v_n\}$ and use the term "simple path" to denote what we have called a path. If $n \geq 2$ and

$v_0 = v_n$, then the path is called a "cycle" (sometimes "circuit"). The *distance* between two points in a graph is the length of the shortest path between them. A graph is called "Eulerian" if there exists a walk that traverses each edge of the graph exactly once, and is called "Hamiltonian" if there exists a path passing through each vertex exactly once. A graph is *connected* if every pair of points is connected by a path. A maximal connected subgraph of a graph is called a "component."

The *degree* (valence) of a vertex of a graph is the number of lines *incident* to that point. In a digraph, the corresponding idea is expressed by "out degree" and "in degree." Clearly,

$$\sum_{v_i \in G} \deg(v_i) = 2q,$$

where q is the number of edges in the graph.

A complete graph on n points, denoted by K_n , is the n -graph containing all possible $\binom{n}{2}$ lines. A *bipartite* (colorable) graph, or *bigraph*, is a graph whose point-set can be partitioned into two subsets, V_1 and V_2 , such that any edge of G connects a point of V_1 with a point of V_2 . A graph is bipartite if and only if all its cycles are of even length. A *complete bigraph*, denoted by $K_{m,n}$, contains all possible lines.

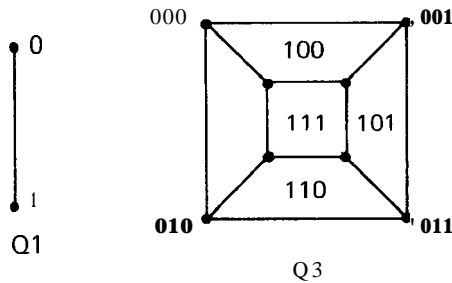


Fig. 8

$Q_{1,3}$ in Fig. 6 is such a graph, with $V_1 = \{v_1, v_2, v_3\}$ and $V_2 = \{h_1, h_2, h_3\}$.

A graph useful in many applications is the “ n -cube” Q_n ; this is a graph on 2^n points where each vertex may be denoted by an n -bit number. Two points of Q_n (such as Q_1 and Q_3) are adjacent whenever their binary representation differs in exactly one place, as in Fig. 8.

A *cut-point* (articulation point) of a component of a graph is a point whose removal disconnects that component, and (analogously) a *bridge* (isthmus) is a line whose removal disconnects a component. A graph is n -connected if n points must be removed to disconnect the graph. A graph is two-connected or *biconnected* if and only if every pair of points of the graph lies on a cycle.

The graphs called “trees” are used in many computer applications. Knuth (1972) defines a *rooted tree* recursively as a finite set T of *nodes* in which

1. One specially designated node is called the “root” of the tree.
2. The remaining nodes are partitioned into $n \geq 0$ disjoint sets T_1, T_2, \dots, T_m (or *subtrees*), each of which is a tree.

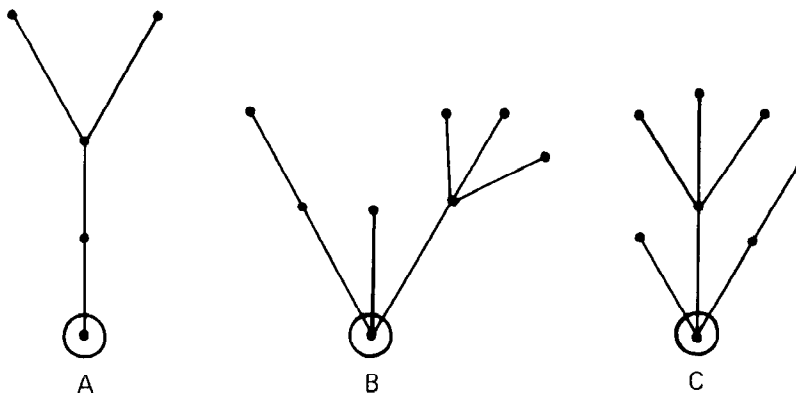


Fig. 9

Fig. 9 illustrates some rooted trees, with the root circled. If there is no point of the tree distinguished as the root, then the tree is called a “free” tree by Knuth, but we will just use the term “tree” here.

It can be proved without too much difficulty that any of the following statements about a graph are equivalent, and hence any one of them may also be used as a definition for a tree. Let G be a graph with n vertices and k edges.

1. G is a tree.
2. G is connected and has no cycles (is *acyclic*).
3. Every two points of G are connected by a unique path.
4. G is connected and has $n - 1$ edges.
5. G is acyclic and has $n - 1$ edges.
6. G is connected, but if any edge of G is removed, then the resulting graph is disconnected.

The word “tree” has lead to much fanciful vocabulary—forest, plantation, leaf, twig, root, palm tree, frond, arborescence, etc., all have technical meanings. In particular, we note that an *end-point*, or *leaf* of a tree is a vertex of degree 1. A *spanning tree* of a graph is a spanning subgraph, which is also a tree. For the application of Kirchoff’s laws (which make use of spanning trees) to flowcharts, see Knuth (1972).

If the order of the subtrees formed by deleting the root is important, i.e., if B and C in Fig. 9 are considered distinct, then the rooted tree is called “ordered.” Ordered trees are important because computer representations of trees necessarily give rise to rooted ordered trees.

For some reason (possibly in analogy to the “family tree”), rooted trees in computer literature are drawn upside down, with the root at the “top.” Computer terminology for tree structures is sexist.

GRAPH THEORY

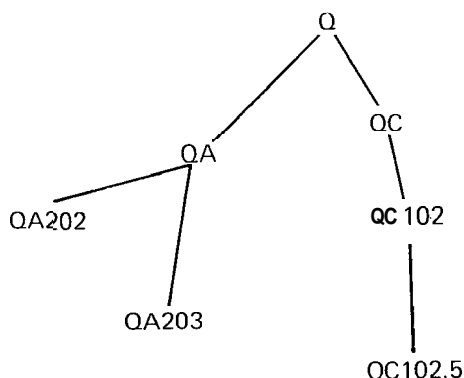


Fig. 10

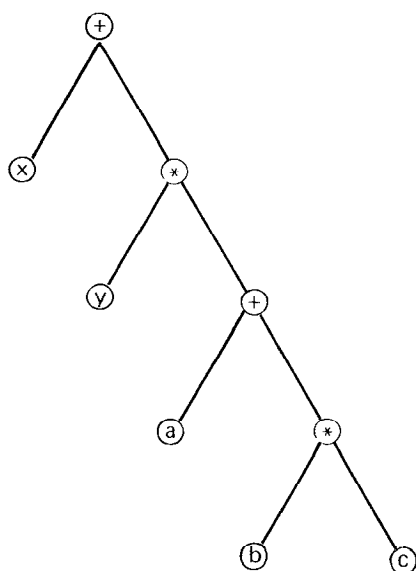


Fig. 11

Every root is said to be a father (mother) to the roots of its subtrees. These roots are called “brothers” (sisters) and “sons” (daughters) of their “fathers” (mothers). The neuter words (parent, child, sibling) do not seem to be used. The root of the entire tree (godhead?) has no father. The egalitarian words “ancestor” and “descendant” are terms that may designate nodes several levels apart in the tree.

A *binary* tree is a rooted-ordered tree, each vertex of which has at most two *subtrees* that are designated left and right. Binary trees arise naturally from a variety of sources. The labeling system of the library classification for books may be thought of as a tree. (See Fig. 10). Algebraic formulas give rise to

tree structures as shown in Fig. 11 for the formula $x + y(a + bc)$.

Many algorithms exist for traversing (searching) a tree systematically. These are of great importance whenever large sets of data are stored in a tree structure and it is required either to find an existing piece of data or to insert a new piece of data appropriately.

The development of graph theoretical algorithms is playing an increasing role in those areas that have come to be called “computational complexity,” and in the analysis of algorithms. On early computers, lack of main storage prevented the solution of graph problems involving many nodes. Moreover, these computers were not fast enough to handle the very large number of computations that are often required for graph-related computations. Today, large main stores and readily available backing stores have alleviated the storage problem but the time problem still remains, particularly for problems where the obvious algorithm requires $n!$ operations.

For example, the problem of deciding whether or not two graphs are isomorphic is a trivial one for the mathematician, who will simply check all $n!$ mappings of one vertex set onto the other. But the time requirements for a problem of this sort quickly get out of hand. If the solution of this problem would require 6 minutes on a computer when $n = 10$, it would require 9 years for $n = 15$ and 300,000 centuries when $n = 20$. In order to overcome this difficulty, computer scientists have, in recent years, invented new approaches to algorithm design and new algorithms for a number of problems that require much less time than $n!$ algorithms. We should note in this connection the pioneering work of A. M. Ostrowski, who (in 1954)—long before the consideration of such problems was fashionable—posed the question: What is the minimal number of multiplications and/or additions necessary to evaluate a polynomial of degree n ? For the current status of this problem see Knuth (1969).

Among the matters of significance in the discovery of fast algorithms for graph problems are the choice of data structure (the use of the adjacency structure instead of the adjacency matrix, for example), depth-first search procedures in tree structures, and recursively dividing the problem into two problems, each one-half the previous size but requiring much less than one-half the computation.

Sometimes the new algorithms are heuristic; i.e., they do not attempt to find the optimal solution (least number of colors in graph coloring, for example) but one that is near optimal.

We now briefly summarize problems for which algorithms have been written and give for each problem the currently best-known upper bound for the time of its execution. We assume our graph is an (n, m) -graph, i.e., one having n points and m edges.

An algorithm is said to require $O(n^p)$ operations if the number of operations divided by n^p is bounded as $n \rightarrow \infty$. Such an algorithm is said to require polynomial time. Among algorithms that are $O(n)$ are those to determine tree isomorphism and those to determine planarity. It is interesting to note that the bound for planarity algorithms has dropped from $O(n^4)$ to $O(n)$ in the past few years. Most connectivity problems—finding cut-points and/or bridges, determining connectivity, biconnectivity, and three-connectivity—are $O(\max(m, n))$; so is the problem for finding a spanning tree of a graph and that of constructing an Eulerian path if it exists. The best algorithm for subtree isomorphism is $O(n^{2.5})$ and that for finding a minimum spanning tree (the edges are given weights and the spanning tree with minimum total weight is required) is $O(\min(n^2, m \log m))$. An algorithm for the isomorphism problem for planar graphs also exists, which is $O(n \log n)$. The best shortest path (edges are weighted) algorithms are $O(n^3)$, as are the algorithms for more general maximum-flow problems.

Some problems are known to require algorithms that are exponential (i.e., there is no p such that the number of operations is $O(n^p)$). For example, the problems of finding all cliques (a clique is a set of vertices, each of which is connected to all others) in a graph, all isomorphisms between two graphs, and all cycles or all paths in a graph are all known to be exponential.

For a rather large class of problems, the upper bound is unknown. These include: finding the largest clique, general graph isomorphism, vertex coloring, and the traveling salesman problem. However, it is known that if an algorithm requiring polynomial time exists for one of them, then so does one for all of them. If any are shown to be exponential, then they all are.

REFERENCES

962. Berge, C. *The Theory Of Graphs and Its Applications*. London: Methuen.
969. Harary, F. *Graph Theory*. Reading, Mass.: Addison-Wesley.
972. Knuth, D. E. *The Art of Computer Programming* [vols. 1 (1968), 2 (1969), 3 (1972)]. Reading, Mass.: Addison-Wesley.

P. J. EBERLEIN

GRAPH ICS. See BIOMEDICINE, COMPUTER GRAPHICS IN; and COMPUTER GRAPHICS.

GROSCH'S LAW

For articles on related subjects see PERFORMANCE OF COMPUTERS; and SUPERCOMPUTERS.

In the late 1940s, Herbert R. J. Grosch formulated what has become known as Grosch's law concerning economies of scale in computers. He proposed that computing power increases as the square of the cost of the computer, or

$$P = KC^2,$$

where P = computing power

K = a constant

c = system cost (either lease price or purchase price)

so that, for example, for twice the money one obtains four times the computing power.

While Grosch himself never published his law, it became part of the oral tradition of the computer industry. It was quoted both seriously and humorously, but eventually gained respectability by being cited in several articles.

Grosch's law has received much attention because of its implications, although it is not susceptible to definitive proof. Several studies lend empirical validity to it, but no one has been able to decide whether it reflects the true value in relation to users' cost or if computer manufacturers use its widespread acceptance to price their goods.

It should be kept in mind that there are limits to the extent that economies of scale can be realized; i.e., there is some point at which the state-of-the-art is reached, beyond which it will cost more proportionally to increase computing power. Within this limit, Grosch's law is a rough guide to determining computing power. However, the calculations of Grosch's law are based on a given point in time and are concerned with the new computers released at that point. The existence of a used-computer market, short- and long-term leases, and third-party leases means that Grosch's law cannot normally be applied to the evaluation of computer purchases or leases.

K. E. KNIGHT AND R. P. CERVENY

GUIDE

GUIDE

For articles on related subjects see **COM-PUTER USER GROUPS**; and **SHARE**.

GUIDE is an international association of users of large-scale IBM computers. It was formed in 1956 as an informal computer users group with members from 44 companies. The name GUIDE originated as an acronym: Guidance of Users of Integrated Data-Processing Equipment.

GUIDE was incorporated as a not-for-profit organization in 1970 under the full name of GUIDE International Corporation. As of 1973, GUIDE was made up of 1,300 member installations. The minimum equipment configuration that a member installation has either installed or on order is System 360 Model 40, or larger, or System 370 Model 135, or larger.

GUIDE has three objectives, as follows:

1. *In relation to its members:* To exchange and disseminate information of mutual interest and value, and to promote sound and professional EDP practices.

2. *In relation to the EDP industry :* To communicate to the IBM Corporation user needs in all technical areas of interest; to review, comment, and exchange information on products and services related to IBM large-scale computers; and to influence the development of computer industry standards.

3. *In relation to the public:* An appropriate involvement regarding public opinion as it relates to the data processing industry.

GUIDE holds general sessions semiannually, usually for three days, in the Spring and Fall. Following a prominent keynote speaker, the individual sessions are usually formal presentations that may take the form of tutorials, user experience panels, workshops, or committee reports. The sub-

ject matter consists of current topics that deal with all facets of the data processing environment. Normally, over 100 individual sessions are conducted at a GUIDE general session, with an attendance of over 3,000 delegates.

Immediately preceding the general sessions, the Division and Group management and the GUIDE working projects meet to perform the major work of GUIDE. The GUIDE project work has become so valuable to its membership that two more meetings per year have been added. These meetings, called Mini-GUIDE meetings, usually consist of over 80 working projects. The objectives of these projects include sharing of information of mutual interest, suggesting enhancements to existing hardware and software, bringing problems with existing hardware and software to the attention of each other and to IBM, promoting effective usage of existing hardware and software products, and/or suggesting the development of future hardware, software, or management techniques.

The roster of GUIDE presidents includes the following persons:

Ed Law, North American Aviation, 1956-1957
Mel Gross, ESSO, 1957-1959

Carl Byham, Southern Railway, 1959-1960

Les Calkins, U.S. Steel, 1960-1963

Otis Simpson, Boeing, 1963-1965

Ottice Tidwell, A T & T, 1945-1967

Earl Althoff, Eastman Kodak, 1967-1969

Herb Seidens ticker, Combustion Engineering, 1969-1971

Garland Cupp, McDonnell-Douglas Automation Co., 1971-1973

Al Burris, Northern Trust Co., 1973-1975

More information can be obtained by writing to GUIDE International Corporation, One Illinois Center, 111 East Wacker Drive, Chicago, Ill. 60601.

B. G. CUPP

HANDSHAKING

For articles on related subjects see **DATA COMMUNICATIONS**; and **TELEPROCESSING SYSTEMS**.

The exchange of predetermined sequences of control signals or control characters between two devices or systems to establish a connection, or to break a connection or exchange data and status information, is commonly referred to as "hand-

shaking." This is best illustrated by means of examples.

Consider first Fig. 1, which shows the sequence of signals on the input-output bus of a small computer when writing a character to a device connected to the bus. The computer first places the device address on the **DATA OUT** lines and raises the **ADDRESS** control line to tell the device that the data on the **DATA OUT** lines is an address. The device recognizes its address and raises the control line **OK**, informing the computer that the device is aware that it has been selected. This causes the computer to

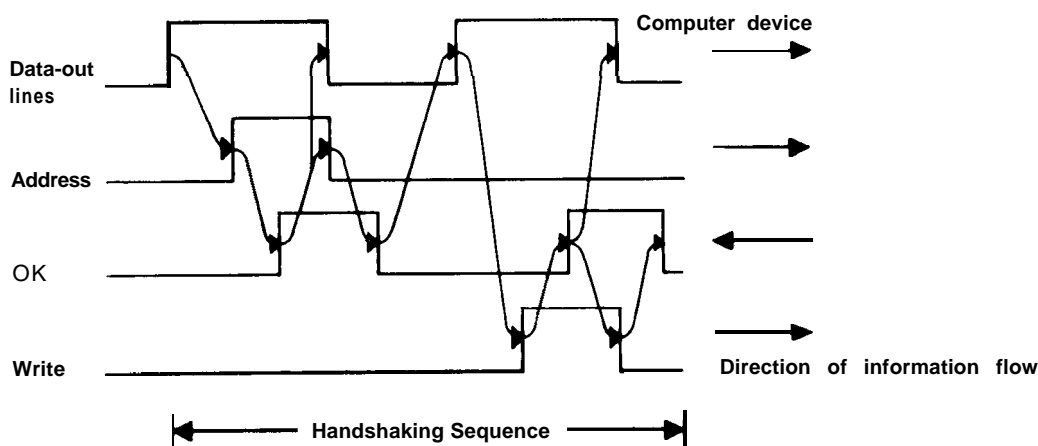


Fig. 1. Example of handshaking sequence. The arrows are used to indicate which control signal causes which response during sequence.

HANDSHAKING

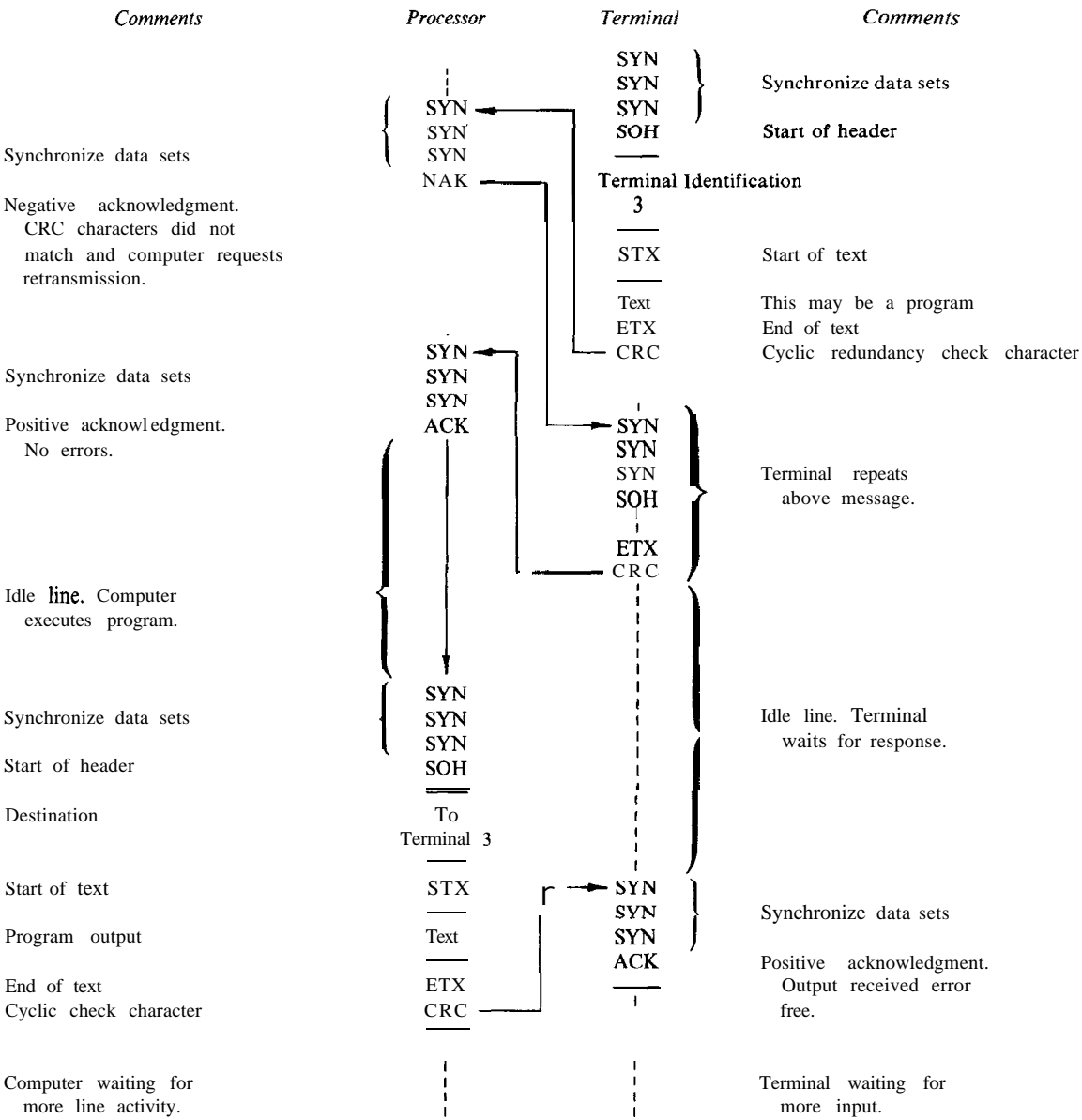


Fig. 2. Handshaking between a computer and a remote batch terminal. The arrows indicate the sequence of line activities.

drop ADDRESS and DATA OUT. The device responds by dropping OK, upon which the computer places the character on the DATA OUT lines and raises the control line WRITE to tell the selected device that the character is on the bus. The device then accepts the character and raises OK, signifying that it has accepted it. The computer then drops DATA OUT and WRITE, which causes OK to go down. This completes

the handshaking sequence for transferring a character from the computer to the device.

Fig. 2 shows an example of handshaking between a computer and a remote batch terminal using synchronous communication. Here the connection is established by a special sequence of control characters (SYN, SOH, STX, etc.). Such handshaking between remote terminals and a computer is often

called "communication protocol," or simply "protocol."

J. S. SOBOLEWSKI

HARD COPY

For articles on related subjects see **INPUT-OUTPUT DEVICES**; and **MACHINE-READABLE FORM**.

For article on related subject see **PAPER TAPE**.

Hard copy is used to describe computer output (strictly speaking only at remote terminals) in either printed or graphical form that can be read directly by humans and handled and filed. This is in contrast to information stored in some magnetic form on tape or disk, or temporarily displayed on a screen, or given by voice, or which requires some special device like a microfilm reader to be read. Such hard copy output may be produced simultaneously with other nonreadable output, which happens, for example, when a magnetic tape or disk file is updated and a printed report is also obtained.

The term is also used when a document that can be read by human beings is produced simultaneously with a machine-readable form of the data. This happens, for example, when a typist simultaneously prepares a typewritten document and a punched-paper tape for input to a computer.

J. NECAS

HARDWARE MONITOR

For article on related subject see **PERFORMANCE MEASUREMENT AND EVALUATION**.

For articles on related terms see **CENTRAL PROCESSING UNIT**; and **CHANNEL**.

A hardware monitor is a device for measuring electrical events (e.g., pulses, voltage levels) in a digital computer. It is useful for gathering data for measurement and evaluation of computer systems, particularly when used in conjunction with software monitoring, a technique using programmed steps

that lead a computer to examine its own internal operation. Most hardware monitors are external general-purpose devices, but in principle they could be built into a computer if economically justifiable. As an example of use, a hardware monitor might be connected to measure the cumulative time the central processor is idle while all I/O channels are busy. Fig. 1 illustrates the elements of a hardware monitor. The various components are discussed below.

GENERAL PROBES. A probe consists of a set of signal sensors designed for minimum interference with the host machine and able to drive relatively long cables so that signals can be picked up from various points physically distant from each other and from the central monitor console.

LOGIC CIRCUITS. The logic circuits accept signals from the general probes and allow logical combinations of the signals (**AND**, **NOR**, **INVERT**, etc.) so that events of interest can be defined.

COUNTERS. A group of counters are used to count the occurrence of various events or to measure the time between events by counting the number of intervening clock pulses.

COMPARATOR PROBES. The comparator probes are similar to the general probes. They are used to sense a number of bits that appear in parallel (e.g., as in an address register).

COMPARATOR. This component provides means for comparing the parallel bits with some preset value at an instant defined by a signal on the strobe line.

DATA TRANSFER REGISTER. The transfer register provides means for passing data directly from the host computer to the magnetic tape record. This register could be combined with the counter functions or with the comparator functions.

In addition to refining the basic functions of gathering data, current development in hardware monitors shows a trend toward emphasizing means for processing **data** during collection and for allowing the host computer and the monitor to alter each other's measurement functions during operation. From the user's viewpoint, the principal differences between hardware and software monitors are:

1. Software monitors can provide more information on cause and effect by relating measured data to the program steps being executed; however, care must be exercised to avoid disruption of time relationships caused by the addition of the measurement programs.

HASHING

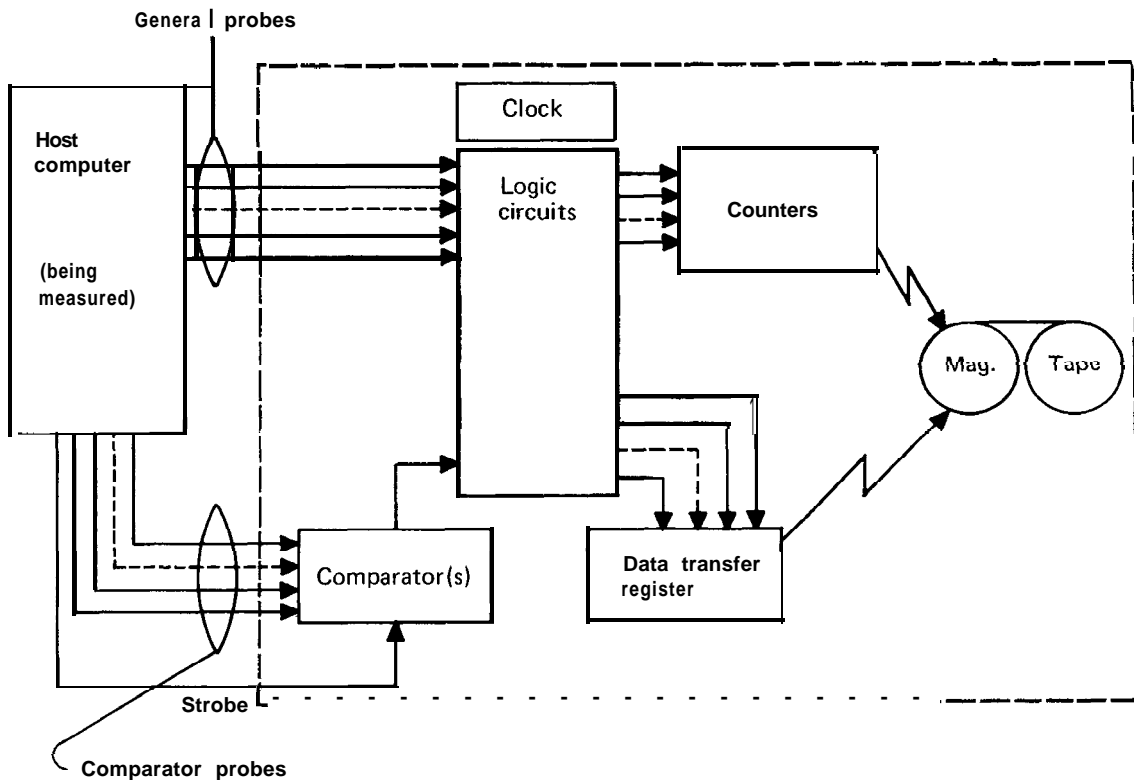


Fig. 1. Elements of a hardware monitor.

2. Hardware monitors only measure electrical **events** at predetermined physical points; hence, it is **more** difficult to relate measurements to program activity. However, with reasonable care, data may be gathered without interfering with the system being measured.

REFERENCES

- 1973. Drummond, M. E. *Evaluation and Measurement Techniques for Digital Computer Systems*. Englewood Cliffs, N. J. : Prentice-Hall, ch. 8, pp. 240-280.
- 1973. Hughes, J., and D. Cronshaw. "On Using a Hardware Monitor as an Intelligent Peripheral," *A CM Performance Evaluation Review*, Vol. 2, No. 4 (December), pp. 3-19.
- 1974. Noe, J. D. "Acquiring and Using a Hardware Monitor," *Datamation* (April), pp. 89-95.

J. D. NOE

HASHING

For articles on related subjects see **SORTING**; and **TABLE LOOKUP**.

For article on related term see **KEY**.

Hashing (or hash coding) is a word coined by computer programmers to describe a general class of operations done to transform one or more fields (usually a key) into a different (usually more compact) arrangement. Probably "hashing" was first coined because it seemed that "hash" was being made out of integral pieces of data. The rationale for hashing is developed more fully in the article on table lookup, dealing with key transformation. The justification for hashing derives from being able to convert naturally occurring, diverse, ill-structured, scattered key fields into compact, easily manipulated fields-usually some numeric, computer-oriented field such as a word or double word, or a computer memory address to facilitate subsequent references. The transformation from the natural field to the

A hash address is only a one-way process, however; the natural field cannot be decoded or reconstructed from the hash. Also, the hashed field may not represent only one unique natural field; many natural fields could hash into the same value.

For example, suppose there is a table of automobile part numbers that are ten numeric characters in length, but there may be no more than 10,000 unique part numbers. In order to contain every possible number, the table would have to allow 10 billion (10¹⁰) positions to handle only 10⁴ possible keys. A scheme can be contrived to transform the original ten-digit key to an integer that will represent the position of that part in a much more compact table.

One simple scheme for hashing is the division-remainder method: Choose a number close to the number of table positions needed. Use that number as a divisor to extract a quotient and a remainder from the dividend (which is the original key). The remainder so obtained is the transformed key. Using 10,000 as the divisor, the transformed key becomes the original key modulo 10,000. Some examples follow.

Original Key (Part Number)	Transformed Key
00 0000 1000	1000
00 0001 0000	0
00 0001 0001	1
00 0001 0099	99
10 0001 0099	99
22 3333 4444	4444
90 0020 0110	110
99 0020 0112	112

The examples in this table were constructed to illustrate the occurrence of duplicate transformed keys. In such schemes, prime divisors are normally used in practice.

Ideally, the hashing scheme would convert the original keys to transformed keys with no duplicates. While schemes can be constructed to minimize collisions ("hash clash") their possibility cannot be eliminated completely and, because of this, the original key must be stored in the table. Further, some scheme must be used to handle duplicate transformed keys.

The examples and discussion in the article on table lookup will further describe methodology and rationale for hashing. Some other techniques in addition to division-remainder are: (1) folding, (2) radix transformation, and (3) digit rearrangement.

Folding consists of splitting the original key into

two or more parts, then adding the parts together (or, sometimes, using the exclusive OR operator). This sum, or some part of it, is then used as the transformed key. For example:

Original key = 20 2152 9396
Splitting and adding: 20 + 2152 + 9396 = 11568
Discard high-order digit to obtain four-digit transformed key of 1568.

Radix transformation involves changing the radix or base of the original key and either discarding excess high-order digits (i.e., digits in excess of the number desired in the key) or extracting some part of the transformed number. For example, an original key of 12345 (base 10) could be considered a base-16 number, and would be transformed as follows:

$$(1 \times 16^4) + (2 \times 16^3) + (3 \times 16^2) + (4 \times 16^1) + (5 \times 16^0) = 74565.$$

The four-digit key would be 4565 by discarding the high-order excess digit(s).

Digit rearrangement consists simply of selecting and shifting digits of the original key. For example, an original key of 1234567 could be transformed to a four-digit key of 6543 by selecting digit positions 3 through 6 and reversing their order.

No one technique is necessarily superior to another in general; however, for specific applications, some may work better than others. The selection of a technique should involve consideration of which technique results in fewest duplicate hash keys.

"Hash" totals are sometimes used for purposes of checking or verification; in this context, hashing has a different purpose than key transformation, inasmuch as the totals may not necessarily be hashed or scrambled. The use of hash totals is for a purpose much like the use of parity bits or self-checking codes for representing characters in digital form on media such as magnetic tapes or punched-paper tapes. When such data is written (or recorded), hash totals are generated and written along with (usually after) the data. Then, when the data is read, the hash totals are recomputed, using the same algorithm, and checked against the ones recorded. If they agree, one can be more certain that the recorded data read is identical to that written and that no bits have been lost or misread.

For example, if a hash total is taken after every

HEURISTICS

five numbers, that hash total could be recorded (written) after the five numbers, thus:

Five data numbers	{	12345
		37654
		8970 1
		00378
		<u>42270</u>
Total		182348

Discard
excess to
obtain "hash"
total

Now, upon reading this data and its hash total during some subsequent process, one could recompute the hash total in the same way and thus verify that it matched the one originally recorded.

C. E. PRICE

HEURISTICS

For article on related subject see **ARTIFICIAL INTELLIGENCE**.

For article on related term see **ALGORITHM**.

The ancient Greek word *heuriskein* means "to find out, to discover." The English adjective "heuristic" and the more recently coined noun "heuristics" came into being via the Latin adjective *heuristicus*. According to the *Random House Dictionary*:

heuristic **adj.** 1. serving to indicate or point out, stimulating interest as a means of furthering investigation. 2. (of a teaching method) encouraging the student to discover for himself. • **n.** 3. a heuristic method or argument.

In the general sense, we talk of the "heuristic power" of a technique, the "heuristics in somebody's reasoning," and so on. Pólya (1954) has written several most entertaining books that do not teach, but do make one realize how to approach problems in mathematics and geometry via heuristic ideas. Also, Hadamard's essay on discovery in mathematics yields an interesting insight—a much too rare phenomenon—into how one of the great mathematicians of all time tackles problems.

How does all this concern us in computing? The reason is simple but its application leads to an area that is completely open-ended. Let us consider, for example, a standard task in programming. We wish to find the roots of a higher-order algebraic equation. There are several methods of approximation that yield the solution with estimatable error bounds. We have the formulas to follow, step by step, and eventually we obtain the results. This is the *algorithmic approach*.

Let us now consider a so-called ill-defined problem, and we have millions of them in everyday life. For example, say we want to balance our household budget by following a program. Although our basic needs are reasonably well known (food, shelter, clothing, medical items, transportation, entertainment, etc.), neither the relative weight of the components nor their unit prices are determinable completely. Also, our needs, desires, and tastes change continually. Our interaction with the environment represents a significant modifying factor. Because this problem is terribly ill-defined, no mathematical technique by itself has a chance to solve it. The computerization of the solution requires all those vague, hard-to-quantify ideas that humans in fact make use of in doing this problem. ("Either I go on vacation or buy that new car. . . . Let's see, how much longer can I drive my old bomb?") The collection of these rules of thumb, sometimes referred to as insight, intuition, or experience with a particular task, represent what computer scientists call "heuristics" (plural noun).

We resort to heuristic programming whenever an algorithmic solution is prohibitively expensive or impossible to follow, or is unavailable. The role of heuristics is to cut down the time and memory requirements of search. On the average, it should result in appreciable savings when programming our budget to satisfy our basic needs. It must be pointed out that heuristic methods are not foolproof; they can fail a certain proportion of the time. (Algorithms are not supposed to fail. . . . However, the fact that a technique is not foolproof does not render it heuristic.)

The larger the range in which a heuristic can be applied, the more powerful it is considered to be. Also, its level of performance is comparable to that of an exhaustive strategy (an algorithm, in fact) or of a random search for a solution.

The following example, originally reported by Simon, should shed some light on the concept under discussion. It is well known that practically any nontrivial game cannot be played by humans *or* by machines algorithmically (because there does not

HIGHER EDUCATION, COMPUTERS IN

xist an algorithm) or exhaustively (because the memory and time requirements far exceed any available ones). The classical example of intellectual games, chess, has been programmed by several groups of researchers. In all these, heuristic ideas occupy a central role in move selection and position valuation. In fact, de Groot and other psychologists have shown that the basic difference between excellent and mediocre players is not in their memory capacity or even in their data processing ability in abstracto. All players analyze practically the same number of board positions, but not always the same ones. Excellent chess players have developed very powerful heuristics for the selection of game continuations to be considered. They may go down to a depth of, say, 20 half-moves along one path and disregard others below a depth of 2 or 3, for reasons of their own.

One of the rather often used heuristics is to have as little freedom of move selection for the opponent as possible. If all other techniques of comparison assign an equal score to two moves considered, a chess expert usually selects the one that restricts the opponent's mobility to a larger degree. This technique, being a heuristic, works most of the time. There was a famous game, however, between two international masters in which the winner used this heuristic to his disadvantage. It has been shown by game analysts that in a particular position, the optimum move (overlooked for the reasons discussed) could have led to an earlier victory. A supplement to this story is that the MATER program by Baylor and Simon (1966), which incorporates the heuristics of fewest-replies, has presented the same particular near-end position and duplicated the mistake made by the international master.

Outlook. Except for some introductory efforts (Waterman, 1970; Findler et al., 1971), present heuristics are all preprogrammed in artificial intelligence projects. In other words, it is not the machine that discovers, selects, and optimizes the rules that play an increasingly important role in many problem-solving programs. Therefore, the performance of these programs is determined by the researcher's experience, insight, and perhaps even luck.

A much more desirable situation would be the one in which the heuristic processes are automated. Learning programs, initially inefficient and possibly even random in their actions, would gradually formulate more and more heuristics on the basis of experience. These heuristics would assume a flexible, or parametric, format so that subsequent optimization

processes could raise the overall level of performance.

REFERENCES

- 1954. Pólya G. *Mathematics and Plausible Reasoning*, vol. I.; *Induction and Analogy in Mathematics*, Vol. II.; *Patterns of Plausible Inference*, Princeton, N.J.; Princeton University Press.
- 1963. Minsky, M. "Steps toward Artificial Intelligence," *Proc. IRE*, Vol. 49, pp. 8-30. Reprinted in Feigenbaum and Feldman (Eds.), *Computer and Thought*. New York: McGraw-Hill.
- 1966. Baylor, G. W., and H. A. Simon. "A Chess Mating Combinations Program," *Proc. SJCC*, vol. 28, pp. 431-447.
- 1970. Waterman, D. A. "Generalization Learning Techniques for Automating the Learning of Heuristics," *Artificial Intelligence*, Vol. 1, pp. 121-170.
- 1971. Findler, N. V., H. Klein, W. Gould, A. Kowal, and J. Menig. "Studies on Decision Making Using The Game of Poker," *Proc. IFIP Congress Book TA-7*, pp. 50-61. Ljubljana, Yugoslavia.

N. V. FINDLER

HIGHER EDUCATION, COMPUTERS IN

For articles on related subjects see EDUCATION IN COMPUTING SCIENCE; and DIGITAL COMPUTERS.

For articles on related terms see ENIAC; MARK I; and SWAC.

Universities as Computer Builders. Much of the early research and development of calculators and (later) stored program digital computers was carried out by university personnel. A few examples are the "large systems of linear algebraic equations" solver at Iowa State College (1937-1942); the Mark I or IBM Automatic Sequence Controlled Calculator at Harvard (1939-1944); the ENIAC (Electronic Numerical Integrator and Computer) at the University of Pennsylvania (1942-1946); the EDVAC (Electronic Discrete Variable Computer) at the University of Pennsylvania (1945-1950); the ORDVAC (Ordnance Variable Automatic Computer) and the ILLIAC (Illinois Au-

HIGHER EDUCATION, COMPUTERS IN

tomatic Computer) at the University of Illinois (1948-1952); the MSUDC (Michigan State University Discrete Computer) at Michigan State; the Whirlwind I at the Massachusetts Institute of Technology (1947-1950); and the SWAC (Standards Western Automatic Computer) at University of California at Los Angeles.

In recent years very few computers have been built by universities. Even the ILLIAC IV developed at the University of Illinois was largely subcontracted to computer manufacturers. However, the University of Illinois continues to be active in computer design with ILLIAC V already under way.

Universities as Computer Users. During the early 1950s universities began to acquire computers for general use in their research activities. Many were brought in to handle large-scale statistical and data processing tasks and were usually augmented by large punched-card processing machine installations. Early calculators included IBM's 602A, 604, and CPC (Card Programmed Calculator). The first stored-program digital computer to be utilized by universities in large numbers was the IBM 650, which was made available for several years (approximately 1955-1963) on a 60% educational allowance.

By early 1960 there were approximately 125 colleges and universities in the United States which had one or more stored-program digital computers. Sixty-five of these had the IBM 650, while the remaining 60 were about equally split between smaller machines and those that were larger than the 650 (Keenan, 1959). Table 1 shows that the growth in the numbers of colleges and universities obtaining computers or acquiring access to computers was rapid. By 1975 practically every college and university campus will, indeed *must*, have access to com-

puter facilities for instruction, research, and/or administration.

The use of computers has permeated nearly every discipline. These widespread uses have pushed computer costs upward from 2% to 5% of the institutions' total budgets in most cases. Table 2 shows estimates of the total expenditures and expenditures per student for fiscal year 1970. Approximately three-fourths of computer funds for the 1970 fiscal year came from these institutions' own funds, with the remaining quarter coming from federal government agencies.

Table 2. Estimated Expenditures for Computing Total and per Student by Highest Level of Degree Offering 1969-70*

Highest Degree Offered	Total Expenditure (Millions)	Expended per Pupil (\$)
Associate	63	33
Bachelor's	26	33
Master's	65	36
Doctorate	318	106
Total	472	63

*John W. Hamblen, 1967.

By fiscal year 1973 the total expenditures for computing by U.S. institutions of higher education was expected to have reached \$600 million and to reach one billion dollars by 1980 (Hamblen, 1967).

DISTRIBUTION OF EXPENDITURES BY FUNCTION. An estimated 30% of the reported expenditures were for on-campus instructional uses of the computing facilities. More than half of these expenditures (18%) were for credit instruction at the institution (see Fig. 1). Approximately 32% was for research and 34% for institutional administration.

In dollar amounts these estimates are roughly \$142 million for instruction, \$151 million for research, and \$161 million for administration. The remaining 4%, or about \$19 million, includes services to other educational institutions (estimated to be 3%) and local industry and government agencies (1%).

STUDIES ON COMPUTERS IN HIGHER EDUCATION. The first comprehensive study to be conducted on the status of computers in higher education was conducted by the National Research Council and published in 1966 (Keenan, 1966). This report is commonly known as the Rosser report, after the study committee chairman, Barkley J. Rosser. This was followed by a publication by the

Table 1. Estimated Number of U.S. Institutions of Higher Education with Access to Computer Facilities

	Total No. of Institutions or Campuses	No. with Access to Computers
1960	2000 (Est.)	125
1965*	2219	707
1967*	2477	980
1970*	2807	1681
1973†	2850 (Est.)	2240

* John W. Hamblen, 1967.

† John W. Hamblen, 1970.

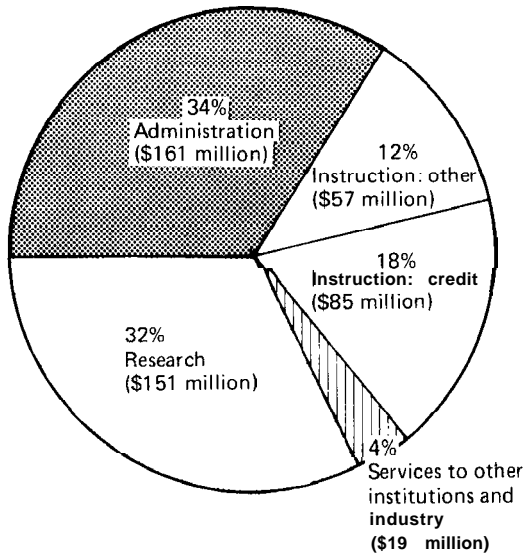


Fig. 1. Estimated distribution of expenditures for instruction, research, and administrative uses of computers in U.S. higher education, 1969-1970.

President's Science Advisory Committee (Pierce, 1967), commonly referred to as the Pierce report, and a publication on the first of three national surveys conducted by the Southern Regional Education Board for the National Science Foundation (Hamblen, 1967). Reports have also been published on studies sponsored by the American Council on Education (Caffrey and Mosmann, 1967) EDUCOM (Mosmann, 1973), and the Rand Corporation (Levien, 1971, 1972).

Associations, Groups, and Societies with Interests in Computers in Higher Education.

ACM/SIGUCC, Special Interest Group on University Computer Centers and ACM/SIGCSE, Special Interest Group on Computer Science Education of the Association for Computing Machinery, 1133 Avenue of the Americas, New York, N.Y. 10036.

AACRAO, American Association of Collegiate Registrars and Admissions Officers, One Dupont Circle, Suite 330, Washington, D.C. 20036.

AEDS, Association for Educational Data Systems, 1201 Sixteenth St. N.W., Washington, D. C. 20036.

CUMREC, College and University Machine Record Conference, 42 Hannah Administration Bldg., Michigan State University, East Lansing, Mich. 48823.

CAUSE, College and University System Exchange, 737 29th St., Boulder, Col. 80302.

EDUCOM, Interuniversity Communications Council, Box 364, Princeton, N. J. 08540.

REFERENCES

1959. Keenan, Thomas. "Survey of University Computing Centers," Computing Center, University of Rochester. Rochester, N.Y.
1966. Keenan, Thomas A. (Ed.). *Digital Computer Needs in Universities and Colleges*. Washington D.C.: National Research Council.
1967. Caffrey, John, and Charles Mosmann. *Computers on Campus*. Washington, D.C.: American Council on Education.
1967. Hamblen, John W. *Computers in Higher Education: Expenditures, Sources of Funds and Utilization for Research and Instruction 1964-65 with Projections for 1968-69*. Atlanta, Ga. : Southern Regional Education Board.
1967. Pierce, John R. (Chairman of Study Committee). *Computers in Higher Education*. Washington, D.C.: White House.
1970. Hamblen, John W. *Inventory of Computers in U.S. Higher Education 1966-67: Utilization and Related Degree Programs*. Government Printing Office, Washington, DC.: Superintendent of Documents, GPO, NS 1.2:C75, Catalog No. NS1.2:C73/1969-70.
1971. Levien, R. *Computers in Instruction: Their Future for Higher Education*. Santa Monica, CA. : Rand Corporation.
1972. Goldstine, Herman H. *The Computer: From Pascal to Von Neumann*. Princeton, N.J.: Princeton University Press.
1972. Levien, R., et al. *The Emerging Technology, Instructional Uses of the Computer in Higher Education*. New York: McGraw-Hill.
1973. Hamblen, John W. *Computer Manpower-Supply and Demand-By States*. St. James, Mo.: Information Systems Consultants.
1973. Mosmann, Charles. *Academic Computers in Service*. San Francisco: Jossey-Bass Publishers.

J. W. HAMBLEN

HOLLERITH, HERMAN

HOLLERITH, HERMAN

For articles on related subjects see **DIGITAL COMPUTERS: History; IBM CARD; and NINETY-COLUMN CARD.**

For article on related biographical information see **WATSON, THOMAS, SR.**

Herman Hollerith (b. Buffalo, N. Y., 1860; d. Washington, D. C., 1929) was the inventor of punched-card data processing and founder of a firm that evolved to become IBM.

For the quarter-century from 1890 to World War I, he had a virtual monopoly on punched-card data processing. He held the foundation patents on the field (U.S. Patents 395 781-395 783) and nearly 50 other United States and foreign patents on basic techniques and equipment. He developed applications of punched-card data processing to many fields of endeavor, including the Census, medical and public health statistics, railroad and public utility accounting, stock and inventory control, and factory cost accounting.

Many basic decisions he made at, or before, the turn of the 20th century persist today. Punched cards today are the size of dollar bills of that era because Hollerith found it economical to buy cabinets and drawers subdivided in that size. The positional coding used on punched cards (Hollerith code) has evolved directly from decisions he made about card design when designing the first column-by-column keypunch for the 1901 Census of Agriculture. Even the practice of IBM and other firms to lease and maintain their own data processing equipment originated in Hollerith's decisions made prior to 1900.

Upon graduation from Columbia University in 1879, Hollerith took a job with the Census, where he became the **protégé** of Colonel John Shaw Billings, an Army surgeon who was also serving as director of the division of vital statistics for the Census. Billings suggested to Hollerith that a good machine to do the purely mechanical work of tabulating population and similar statistics was badly needed and that a technique of using cards with the description of each individual punched into them was a good approach to the problem. Intrigued by this suggestion, Hollerith made a study of the problem and determined to his own satisfaction that it was feasible.

In 1882 Hollerith followed General Francis Walker from the Census to M.I.T., where he became an instructor in mechanical engineering. While there he worked hard on his "Census machine" invention, concentrating initially upon a variant of an earlier

machine developed by Colonel Charles W. Seaton, chief clerk for the 1870 Census. This prototype had used a player-piano roll type of feed mechanism rather than individual cards.

By the end of his first year at M.I.T., Hollerith decided that his true vocation was invention. He returned to Washington and secured a position with the Patent Office to learn the arts of invention and patent protection. After a year, he left the Patent Office and set up shop as a "Solicitor and Expert on Patents" to earn his living while he gave his primary attention to invention. Soon he applied for several patents; included among them was the first application for the foundation patents on punched-card data processing.

Considering this to be the most promising of his inventions, he concentrated upon it and developed the experimental test systems used for vital statistics tabulations in Baltimore, New Jersey, and New York City. During this period, his system evolved from a simple machine with cards punched by a conductor's ticket punch to a complete system. This system included a pantographlike punch, a tabulating machine with a large number of clocklike counters (each capable of counting up to 10,000 occurrences), and a simple, electrically actuated sorting box for classifying and grouping cards in accordance with the categories punched into them.

In 1889 his system was installed in the Army Surgeon General's office to handle Army medical statistics. A description of this system and his plans for the Census was accepted by Columbia as a doctoral dissertation, and he was awarded a Ph.D. "for achievement" in 1890. Also in 1889, a comparative test made of the Hollerith and two competitive systems caused the Hollerith system to be chosen for use in the 1890 Census. Austria, Canada, Italy, Norway, and Russia were soon investigating and adopting Hollerith equipment for their population censuses. These early systems could only tally, not add or accumulate, totals one at a time.

Shortly after 1900, Hollerith began developing a second generation of his equipment. A new type of card design arranged numeric information in columns and permitted development of a simple, new kind of keypunch, an automatic-feed card sorter, and an automatic-feed tabulator of vastly improved performance. These new systems could accumulate numbers of any size, and thus were obviously applicable to many situations other than census and similar statistical work. Hollerith soon spread their use to an amazing variety of industries. They even went overseas with the American Expeditionary Forces in World War I.

About 1905 the management in the Census Bureau began to object to Hollerith's profits and sponsored alternative developments designed to break his monopoly. These competitive systems were widely adopted, once Hollerith's fundamental patents expired, and often led the data processing industry into new developments. Because this competition resulted in a need for increased capitalization, Hollerith sold his patent and proprietary rights to a holding company in 1912. This relieved him of day-by-day management chores and he became a highly paid consultant. Before long, Thomas J. Watson, Sr., was brought in to head Hollerith's old company, but Watson's commercialism and Hollerith's devotion to purely inventive objectives caused dissension. Watson's interests prevailed, and Hollerith's contributions and achievements were soon absorbed in the greater representative image of IBM.

REFERENCE

Hollerith, Virginia "Biographical Sketch of Herman Hollerith," *ISIS*, Vol. 62, No. 210, pp. 69-78.

W. F. LUEBBERT

HONEYWELL. See **MANUFACTURERS, COMPUTER.**

HOSPITAL INFORMATION SYSTEMS

For articles on related subjects see **INFORMATION SYSTEMS; MANAGEMENT INFORMATION SYSTEMS;** and **MEDICAL APPLICATIONS.**

The automation of health-care delivery is variously referred to as the development of hospital, health, or medical information systems. These systems have in common a high level of on-line operation and a file structure based on a patient-oriented record.

The physician and nurse are central to a hospital information system (HIS), since most activities in a medical environment are based on a physician's order. Such an order may activate tasks in many areas: the pharmacy, the clinical laboratory, the

kitchen, accounting, etc. It is obvious that communication capability is a primary requirement. It has been shown that 20 to 30% of the nursing staff time, and a similar percentage of total hospital costs, are related to information processing. Beyond simple communication services, a HIS system may be used in a pharmacy for the preparation of medication schedules, drug compatibility, and interaction checking. Clinical laboratory tasks such as blood collection schedules, test scheduling and instrument monitoring, result normalization based on standard test samples, and outcome posting, with flags indicating abnormal or out-of-range results may be produced by modules of an HIS. Other tasks can include the generation of patient care plans, diet planning, inventory maintenance, scheduling of staff, etc..



Fig. 1. Cathode-ray tube terminal displays clinic scheduling information coming from the Honeywell central computer at Children's Hospital Medical Center, Boston.

For the tasks mentioned above, only a fairly transitory record has to be kept on the patient. In order to impact the medical care to a greater extent, a computerized medical patient record will have to be kept. The entry of physicians' and nurses' observations on the patient, combined with formal categorizations such as problem statements and diagnoses, will allow summarization of a patient's history and progress. Given adequate decision aiding tools, the possibility for useful interaction for the physician emerges. Without such feedback, the utility to the

HOST SYSTEM

physician of using a computer entry device, no matter how sophisticated, is at best negligible.

Much data entry into actual systems is performed by the nursing staff and ward clerks. A long-term medical record implies extension of the HIS into the outpatient, community pharmacy, and private physician areas. A well-organized medical record may distribute the responsibility for health-care delivery among physicians, nurses, specialists, and medical paraprofessionals, since less will depend on the memory of the physician in attendance. The goal of a complete and integrated medical record is, however, quite elusive, unless supported through forms of comprehensive health insurance.

Systems providing major or minor portions of the tasks outlined above have been provided in a variety of forms. Entry devices may be readers of prepunched cards, large function keyboards, conventional typewriter terminals, as well as CRT terminals. The most successful methods have utilized lightpen or finger-touch selection from CRT-presented choice lists. In a well-structured hierarchy of choices, any specific order may be entered with three to seven screen loads. A major problem has been the poor quality of output presentation. Excessive quantities of information printed, sometimes noisily, on identical white sheets using lines of uniform uppercase-only characters with unpleasant fonts have made the tasks of searching for relevant information in the record harder instead of easier. A more graphic presentation of the patient's state, as now frequently provided by the nursing staff, may make HIS output more acceptable.

The computer systems themselves range from minicomputers doing fairly isolated tasks to shared computer utilities. Conceptually, systems may be divided into modular systems, which perceive independent development of applications with subsequent linkages, and total systems, which attempt to cover all services within one scheme. Difficulties with modular development have been due to program incompatibilities and lack of intercomputer communication standards, which have prevented the integration of individually successful application modules from many sources into a cohesive network. The total systems have been associated with high initial costs and long delays before they could achieve productive operation. The delays tend to have the effect that the hardware is no longer optimal when the system becomes operational. Total systems have generally employed large central facilities, and here the high demands on file and communication capabilities, which are required to provide CRT terminal interaction with response times

on the order of a second, have caused additional problems.

The increasing demands on health-care quantity, quality, distribution, and cost control will have the effect that, even with the problems listed above, continued attempts at HIS implementations will be made. With better understanding of system requirements, a suitable system architecture can be developed. With imaginative human interface engineering, the acceptability of these systems can be increased. With the inclusion of a complete range of services, economic benefits may accrue which will make the concept of an HIS truly viable.

G. WIEDERHOLD

HOST SYSTEM

For articles on related subjects see **MICRO-PROGRAMMING**; and **MULTIPROGRAMMING**.

A host system or a host computer is the physical system that interprets a program. The program is written on a "logical machine", which is usually not the same as the physical machine (host system). These differences arise because the physical system either does not possess or does not allocate all features or resources directly requested by the logical machine (program). The distinction between host system and logical system is especially notable in two areas: multiprogramming systems and microprogrammed (emulated) systems.

In multiprogramming systems the host system is responsible for allocating storage and I/O resources to each of the logical machines (i.e., in effect, active programs), which are usually called "virtual" machines, as they are required. This allows a number of virtual machines to share the physical resources without logical conflict (i.e., without any programmer intervention in the source programs) and at the same time more effectively use the resources of the physical host system.

In microprogrammed systems the notion of host system applies to the physical machine that interprets (emulates) the programs written in other machine languages. The machine being emulated by the host machine is said to be the "image" machine (sometimes the term "virtual" is also used to describe this situation).

M.J. FLYNN

HUMANITIES APPLICATIONS

For articles on related subjects see **ARTS APPLICATIONS; INFORMATION RETRIEVAL;** and **SOCIAL SCIENCE APPLICATIONS.**

For articles on related terms see **COMPUTER GRAPHICS; DATA BANK; HEURISTICS; PATTERN RECOGNITION;** and **SYNTAX, SEMANTICS AND PRAGMATICS.**

The use of the computer in the humanities entails the preparation of information banks, which include data and rules specifying procedures to be used for applications involving either pattern recognition or pattern generation, or both.

Information Banks in the Humanities.

The preparation of information banks implies preliminary decisions as to formatting and editing. Because many of the traditional categories in the humanities are not rigorously defined, investigators wishing to use such categories often pre-edit the text so as to be able to retrieve information concerning, for example, syntactic or semantic patterns. Thus, at this stage of the use of the computer in the humanities, text formatting and editing sometimes makes use of poorly defined elements and weakly defined or, really, nonexistent models. For example, categories such as image or texture are not sufficiently defined to permit specification by rule for their recognition by the computer; images must be identified in advance by the human being using the computer.

As empirical data gathering, model building, and model testing facilitated by the computer grow, the need for formatting information so as to anticipate the theory upon which the conclusions are based should decline. Rather, it should be possible to "bank" information in its conventional form and then provide rules for sorting the elements in the bank into desired categories. At that point, preparation of information banks will clearly be a step prior to, and separate from, pattern recognition and pattern generation applications in the humanities. At present, although the functional distinction is relatively clear, the methodological distinction is not.

Types of information banks currently used in computer-based applications in the humanities include bibliographies, texts, dictionaries, historical records, and rules for combining discrete elements of whatever artifact is being generated. For the value of bibliographical data for information retrieval, the reader is referred to the article on that subject. The

annual Shakespeare bibliography and the U.S. Modern Language Association's annual bibliography are examples of major data banks of this sort for the humanities. General-purpose dictionaries (e. g., Webster's, Random House) that are available in computer-accessible form are used by humanists (Olney and Ramsey, 1970), for example, to search for literary themes; in addition, humanists have prepared special-purpose dictionaries (e.g., The Old English Dictionary) for research related to their special fields of interest (Cameron, Frank, and Leyerle, 1970).

An example of an information bank consisting of historical records is the 11-volume, 8,000-page, 3-million word collection on the London stage, 1660-1800, an

exhaustive calendar of plays, entertainments, afterpieces, dancing, and singing, together with casts, box receipts, advertising, contemporary comment, and all available information about scenery, theatre construction, costuming, audiences, management, and production, compiled from the playbills, newspapers, and theatrical diaries of the period. (Daland and Schneider, 1971.)

If this material were not available in computer-accessible form, the scholar who

wished to determine how many times actor X and actress Y performed in the same play together during their careers might find it necessary to scan a period of 15 to 20 years (possibly 800 to 1000 pages) to exhaust all the possibilities of intersection, and yet the list of joint performances might not fill a page. (Daland and Schneider, 1971.)

Historical records of all sorts, ranging from voting records to land use records, are being put into computer-accessible form for scholarly research (see Dollar and Jensen, 1971; *Historical Methods*; Swierenga, 1970).

Information banks of recorded speech are of interest to humanists because research based upon such banks may provide useful analytical categories for the auditory components of language and literature. Speech information banks require formidable quantities of storage (a 2,400-foot magnetic tape will hold 5 to 10 minutes of digitized speech), and techniques for coping with the data are likewise complex.

HUMANITIES APPLICATIONS

Information banks comprising rules are currently of importance for pattern generation and will become increasingly significant for pattern recognition as the categories being recognized become ever more amenable to description through a precisely defined procedure. An example of such a bank of rules would be those necessary to enable the computer to approximate the form of a sonnet; e.g., number of lines, number of syllables per line, stress patterns within the line. Some rules (e.g., number of lines) would be obligatory, whereas options would be available as to the selection of some others; e.g., within certain metrical contexts an anapest (UU/) might be substituted for an iamb (U/).

Pattern Recognition. Pattern recognition based upon data banks consisting of bibliographies is described in the article on information retrieval. Pattern recognition based upon historical records is analogous to that involved in bibliographies-i.e., searching for either a particular type of record (e.g., cast listings in Daland and Schneider's "London Stage") or a specified word (e.g., the name of a particular actress in the "London Stage").

Pattern recognition for which the data banks are texts in machine-readable form-dictionaries, and indeed sometimes bibliographies and historical records-might be thought of as falling into three divisions: (1) recognition, using categories that derive from standard graphic conventions; (2) recognition, using categories that derive from traditional approaches to language and literature; (3) recognition, using categories that derive from theories or models concerning the description of discrete and continuous events (from the mathematical sciences).

Examples of categories that derive from standard graphic conventions include characters, words, sentences, and strings longer than sentences. Occurrences of single characters or short strings of characters are of interest to humanists concerned with manuscript *stemma* or with various editions of a given text. The derivation of one manuscript from another is of interest for two reasons: One is that the scholar would like to find the manuscript that is either the original or closest to the original manuscript of the text in question; the other is that shifts in spelling provide data for studies in the phonological history of a given language (Mullen, 1971). Comparisons of editions of a given text are undertaken in order to list the variants among them, and thus to arrive at either an edition that seems best to reflect the presumed original or to arrive at an edition that at least exhibits internal consistency (Cabaniss, 1970).

The presence or absence of specific characters in a text may also provide guides to the style of the author. Insofar as linguistics is considered part of the humanities, the character and short strings of characters are important guides to morphology; e.g., the discrimination of roots, affixes, and plural morphs. Such morphological patterns are of obvious importance for the study of foreign languages, traditionally considered humanistic disciplines.

The humanities have used the word as a category most centrally in indexes and concordances. An index is generally taken to be a listing of the word together with locations of its occurrence in the text. A concordance is a listing of the word together with a specific quantity of context for each of its occurrences in the text. For the production of research aids such as these, the computer is now regarded as an indispensable tool by scholars in the humanities. Words are also used to provide stylistic clues, which in turn are sometimes used for author identification (McKinnon and Webster, 1969). For example, the study by Mosteller and Wallace (1964) of the Federalist papers indicated that it is possible to discriminate between those written by Hamilton and those written by Madison on the basis of certain function words (e.g., articles, prepositions, conjunctions) that occur in the known writings of one author at a frequency significantly different from that in the known writings of the other author. Manuscript and text collation also depend heavily upon words, as well as characters, for purposes of comparison (Spencer, 1972).

Sentences and strings longer than sentences, such as paragraphs and chapters, are categories that derive from standard graphic conventions and may provide guides to various components of style. Decisions as to the units by which these strings are measured may well affect their usefulness for a particular problem. For example, although Mosteller and Wallace (1964) could find no significant difference among the sentence lengths of Madison and Hamilton when measured in words, Robert Wachal has suggested in a doctoral dissertation (on Linguistic Evidence, Statistical Inference, and Disputed Authorship) that there may be a significant difference if those sentences are measured in terms of syllables. Whatever the measure, this approach to pattern recognition can be replicated by other scholars as long as the unit is defined according to standard graphic conventions. Perhaps that is the salient point about pattern recognition based upon categories that derive from standard graphic conventions. The conventions can be clearly defined,

departures from the conventions can be clearly specified, and experimentation is thus replicable.

It is with the next group of categories, those traditionally used for studies in language and literature, that replicability and reliability become a serious problem. Among the categories that fit into this general class are those drawn from phonology, **syntactics**, and semantics, as well as terms that refer to readability and terms such as unity and texture which refer to other terms such as structure and content, etc.

Phonology is concerned with the sound of the spoken language, and one might assume that there are categories related to phonology analogous to those, such as characters and words, which relate to the graphic representation of the language. It is indeed possible to produce information banks of digitized speech and subsequently to make statements concerning acoustic properties such as frequency and intensity (amplitude). Unfortunately, the quantity of storage demanded by even short strings of digitized speech has thus far prevented storing literary or linguistic strings in sufficient length to permit the development or discovery of categories arising directly from the acoustic data-categories that would facilitate discussion of metrics, rhythmic patterns, or rhyming patterns. If, given acoustic parameters, the latter categories no longer seem relevant because they lack equivalent precision, then acoustic patterns might be used to distinguish one utterance, or one poem, or one prose work from another.

Current efforts to map the short strings of acoustic signals that are available onto graphic conventions such as characters, combinations of **characters**, or words are extremely complicated pattern recognition problems. Syllable recognition can now be achieved approximately 80% of the time, but **dentification** of word boundaries is much less **successful**. (For further discussion of the pattern **recognition** problem in speech see the article on speech recognition.)

In lieu of direct input and analysis of acoustic signals, phonology has been studied on the basis of **human** graphic transcriptions of the auditory **component** of language and literature. Although linguists are extensively trained and become skilled at making such transcriptions, recording variabilities do **introduce** very serious problems as far as reliable **repliability** is concerned. Nonetheless, mappings from **phonetic** transcriptions onto graphic conventions, and vice versa, are of interest to humanists wishing to describe patterns of rhyme, of rhythm, and of auditory phenomena.

Syntactic categories are used for stylistic discrimination applied to author identification or to changes in the style of a given author over time. The syntactic categories used depend upon traditional intuitions concerning parts of speech, plus intuitions concerning the text being examined, or upon a particular linguistic model, e.g., transformational or phrase-structure, (See Burton, 1970; Cluett, 1971; Green, 1971; **Koster**, 1970; Milic, 1970.) The use of the computer to locate syntactic patterns for which the syntactic units are defined on the basis of traditional intuitions or even on the basis of various models usually implies pre-editing of the text. Because the syntactic categories used in such **pre-editing** are not sufficiently rigorously defined, efforts to replicate the use of a particular set of categories may not succeed. Insofar as a given scholar is consistent in his use of a set of categories, he may well be able to make statements concerning use of the different categories in the text he is examining. But if another scholar questions his consistency and attempts to replicate his work, difficulties may ensue because the categories themselves are somewhat amorphous (Koster, 1970; Milic, 1970). **Computer-based** parsers have thus far not been used for extended studies of syntactic patterns in literary or other texts, presumably because such parsers tend to provide so many possible syntactic readings for a given sentence as to make analysis of such readings for extensive quantities of text extremely difficult and time consuming. Probabilistic parsers, which give the single best (most probable) parsing for a given sentence, seem most promising for this application (**Stolz** et al., 1965).

Semantics, which is just coming into its own in linguistics, has long been of interest to humanists. Categories such as texture, theme, and tone presumably refer to semantic implications of words and to the relations among those words in a text. Computer-based efforts to utilize semantics in literary analysis have depended upon interplay between words in the text and words in standard reference works such as *Roget's International Thesaurus* or portions of the Oxford English Dictionary (Sedelow and Sedelow, 1966, 1967, 1969; Spolsky, 1970). For literary analysis, large stores of semantic information are necessary; hence artificial intelligence efforts to cope with semantics (Quillian, 1969; Winograd, 1972)—efforts that deal with very restricted universes of discourse—are not thus far viable for literary analysis. Rather, computer-based work in the humanities rely on reference works (dictionaries and thesauri) reflecting general use of the language as both validated and modified through

HUMANITIES APPLICATIONS

time. The difficulty in using such reference works is that their own structures—particularly of thesauri—are not sufficiently characterized so as to ensure that the research scholar knows the biases they introduce into his results. If such references are to be used with assurance, research on modes of characterizing these reference works is necessary; some such research is in progress (Olney and Ramsey, 1970; Revard, 1969; Sedelow and Sedelow, 1969). When semantic networks both external and internal to texts can be more adequately characterized, categories such as texture and tone may either assume new meaning or completely disappear as viable indicators of literary style and structure.

Other traditional categories such as readability and dramatic climax depend either upon categories (e.g., syntactic and semantic) that fall into this general area of categories traditional to language and literature or upon categories such as word length which depend upon standard graphic conventions, or both.

The third major category area important to pattern recognition in the humanities is that derived from the mathematical sciences, particularly statistics and probability, information theory, and analysis (especially relevant to acoustical phenomena). Models in the mathematical sciences can, of course, be rigorously defined and thus tested extensively against data. Because of its quantity, humanistic verbal data is especially appealing to mathematical model builders who in the past have sometimes been dependent upon the restricted number of subjects convenient to social science experiments or to **restricted** instances of any given case. On the other hand, the application of mathematical and statistical models to natural language data is often hampered by the very quantity of data. Contingency tables, transition matrices, and other structures requisite to mathematical and statistical models quickly become so large that they exceed the capacity of even very large digital computers. Efforts to find models that fit data more adequately (e.g., nonparametric statistical models) as well as to find ways of managing data so as to increase data computability are likely to increase dramatically during the coming years as more data becomes available in computer-accessible form and as more model builders become aware of this large storehouse of data available for model testing.

Pattern Generation. In many respects, computer-based pattern generation represents the obverse of pattern recognition. At first blush it might seem to be more difficult because it demands rules

specifying the way elements being generated are to be combined. This would seem to be more complicated than, for example, a search through a text for occurrences of specified words. It is probably true that the most elementary generation of poetry is more complicated than elementary searches for words. But as pattern recognition efforts become more comprehensive so as to include, for example, semantic relationships, the pattern recognition task then becomes more complicated than elementary generation tasks. An analog might be the greater ease with which computer-based transformational generative grammars are written than are **computer**-based transformational parsers. It is easier to generate acceptable sentences than to attempt to parse the infinite variety generated by man.

Pattern generation, like recognition, uses information banks such as dictionaries, thesauri, bibliographical information, and banks of rules based upon models or assumptions appropriate to the artifact being generated. For example, generation of haiku poetry would entail specification of the number of lines, the length of lines, acceptable sequencing of syntactic classes, the vocabulary that falls into the general type of vocabulary appropriate to haiku, and some appropriate semantic relationships among the words. Specification of the latter has been minimal in poetry generated to date. For haiku, a verse form for which semantics is relatively obscure anyway, this specification is not so vital, and in fact, in some contemporary poetry, semantic relationships among words are certainly oblique and often distant. The casual reader may not see a great deal of difference between “a great king packed in an acorn” written by a human and “dance, oh life, like a silent tumbleweed!” written by a machine (Boroff, 1971).

Inasmuch as sophisticated pattern generation of language and literature is extremely complicated, in large measure because of semantics, computer-based pattern generation in the humanities will doubtless be used in the foreseeable future for testing models or parts of models of language and literature. The computer may never write like Shakespeare, but it may be used to identify precisely some aspects of the nature of Shakespeare’s writing.

Although computer graphics represents pattern generation in a somewhat different sense, its use in the humanities as a heuristic, or aid to insight, should be mentioned. Graphics can help reveal patterns in the structure of a literary or other text. Thus, graphics can show which themes or ideas tend to cluster together in a text and which never appear together, which metrical patterns tend to dominate

and where; in short, graphics can be used to represent those elements of the text that have been specified and identified by the computer. Most often, computer graphics provide visual representations of the results obtained by the application of mathematical and statistical models to data.

Other Applications. One of the major areas of the humanities for which the scope of this article does not permit coverage is instruction. Computer-assisted instruction is being used in the teaching of foreign languages, and there is a growing effort to use it in teaching composition. For a detailed presentation of current work in computer-assisted instruction, the reader is referred to the article on that subject.

REFERENCES

1964. Mosteller, Frederick, and David Wallace. *Inferences and Pispu ted Authorship: The Federal-ist*. Reading, Mass. : Addison-Wesley.
1965. Stolz, Walter S., Percy H. Tannenbaum, and Frederick V. Carstenson. "A Stochastic Approach to the Grammatical Coding of English," *Communications Of the ACM*, Vol. 8, No. 6 (June), pp. 399-405.
1966. Sedelow, Sally, and Walter A. Sedelow, Jr. "A Preface to Computational Stylistics," in Jacob Leed (Ed.), *Computer and Literary Style*. Kent, Ohio: Kent State University Press, pp. 1-13.
1967. Sedelow, Sally, and Walter A. Sedelow, Jr. "Stylistic Analysis," in Harold Borko (Ed.), *Automated Language Processing: The State of the Art*. New York: John Wiley, pp. 181-213.
1969. McKinnon, Alistair, and Roger Webster. "A Method of 'Author' Identification," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 2, No. 1 (March) pp. 19-23.
1969. Quillian, Ross M. "The Teachable Language Comprehender: A Simulation Program and Theory of Language," *Communications Of the ACM*, Vol. 12, No. 8, pp. 459-476.
1969. Revard, Carter. "On the Computability of Certain Monsters in Noah's Ark: Using Computers to Study Webster's New Collegiate Dictionary and the Merriam Webster Pocket Dictionary," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 2, No. 2 (August), pp. 446-461.
1969. Sedelow, Sally, and Walter A. Sedelow, Jr. "Categories and Procedures for Content Analysis in the Humanities," in George Gerbner et al. (Eds.), *The Analysis of Communication Con-*
- tent*. New York: John Wiley, pp. 487-499.
1970. Burton, Dolores M. "Aspects of Word Order and Two Plays of Shakespeare," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 3, No. 1 (January), pp. 34-39.
1970. Cabaniss, Margaret Scanlon. "Using the Computer for Text Collation," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 3, No. 1 (January), pp. 1-33.
1970. Cameron, Angus, Roberta Frank, and John Leyerle, (Eds.). *Computers and Old English Concordances*. Toronto: University of Toronto Press.
1970. Koster, Patricia. "Words and Numbers: A Quantitative Approach to Swift and Some Understrappers," *Computers and the Humanities*, Vol. 4, No. 5 (May), pp. 289-303.
1970. Milic, Louis T. "Comment on Mrs. Koster's Article," *Computers and the Humanities*, Vol. 4, No. 5 (May), pp. 304-306.
1970. Olney, John, and Donald Ramsey. "From Machine-Readable Dictionaries to a Lexicon Tester: Progress, Plans, and an Offer." *Computer Studies in the Humanities and Verbal Behavior*, Vol. 3, No. 4 (November).
1970. Spolsky, Ellen. "Computer-Assisted Semantic Analysis of Poe try," *Computer Studies in the Humanities and Verbal Behavior*, Vol. 3, No. 3 (October), pp. 163-168.
1970. Swierenga, Robert P. *Quantification in American History; Theory and Research*. New York: Atheneum.
1971. Borroff, Marie. "Creativity, Poetic Language and the Computer," *The Yule Review*, Vol. 60, No. 4 (June), pp. 481-513.
1971. Cluett, Robert. "Style, Precept, Personality: A Test Case (Thomas Spratt, 1635-1713)," *Computers and the Humanities*, Vol. 5, No. 5 (May), pp. 257-278.
1971. Daland, Will, and Ben R. Schneider, Jr. "The 'London Stage' Information Bank," *Computers and the Humanities*, Vol. 5, No. 4 (March), pp. 209-214.
1971. Dollar, C., and R. Jensen. *Historian's Guide to Statistics*. New York: Holt, Rinehart and Winston.
1971. Green, Donald C. "Formulas and Syntax in Old English Poetry: A Computer Study," *Computers and the Humanities*, Vol. 6, No. 2 (November), pp. 85-94.
1971. Mullen, Karen A. "Using the Computer to Identify Differences Among Text Variants," *Computers and the Humanities*, Vol. 5, No. 4 (March), pp. 193-202.

HYBRID COMPUTERS

1972. Spencer, Christopher. "Shakespeare's *Merchant of Venice* in Sixty-Three Editions," *Studies in Bibliography*, Vol. 25, pp. 89-106.
1972. Winograd, Terry, *Understanding Natural Language*. New York: Academic Press.
- 1967-present. *Historical Methods Newsletter*. Department of History, University of Pittsburgh, Pittsburgh, Pennsylvania.

S. A. SEDELOW

HYBRID COMPUTERS

For articles on related subjects see **ANALOG COMPUTERS**; **DIGITAL TO ANALOG CONVERTERS**; **SIMULATION**; and **SPECIAL PURPOSE COMPUTERS**.

For articles on related terms see **PARALLEL PROCESSING**; and **PROBLEM-ORIENTED LANGUAGES**.

Overview. The history of hybrid computers and computation has been brief but remarkably active. The impetus for hybrid computation was provided by the important dynamic system simulation and optimization problems arising from programs of the National Aeronautics and Space Administration (NASA) and Department of Defense (DOD) during the late 1950s to middle 1960s. The speed of analog computers in solving differential equations, coupled with the high precision and programmed control capability of digital computers, proved to be an ideal adjunct to the dynamic system design process.

The hybrid computers of the late 1950s consisted chiefly of ad hoc combinations of standard analog and digital computers connected primarily through high-speed digital-to-analog and analog-to-digital converters. The principal justification for the connection was for the digital computer to compute one or more complicated functions of several variables (e.g., aerodynamic drag as a function of velocity and altitude). In this particular form, the digital computer appeared to the analog computers as one or more computational elements.

As small, inexpensive digital computers became available during the early 1960s, they were integrated into large analog computers to automate the detailed and time-consuming machine diagnostic and problem setup and checkout tasks. These computers, however, were basically analog computers

with some digital automation. With the increasing awareness of digital computation techniques and the development of higher-level problem-oriented languages, the middle 1960s saw the development of several commercial product lines of integrated hybrid computers with sophisticated support software. Generally, these integrated hybrid computers provided extensive communication and control paths between the constituent analog and digital computers. The digital computer could simulate one or more functional elements for the analog program and still control the computational sequence, with the analog program appearing as a subroutine to the digital computer program. These computers had real impact on personnel previously oriented solely to the analog computer, and attracted the interest of a number of digital computer enthusiasts as well.

Hybrid computers had significant impact on the success of the NASA Apollo program. Most of the Apollo subsystems were simulated extensively during the design process, and many actual subsystems were evaluated and refined in extensive hybrid computer-based simulations involving both real and simulated hardware. Hybrid simulations were also employed extensively in the training of astronauts and support personnel. During the mid-1960s, hybrid computers were also employed in developing the emerging science of digital process control. Much of the now established success of digital computer control in all process industries (paper, petrochemical, nuclear power, etc.) is due to the testing and refinement of early designs through hybrid simulation.

Most of these successful applications of hybrid computers, however, represented straightforward extensions of analog computer simulation techniques. For a time it appeared as though a new discipline of general-purpose hybrid computation was about to emerge. Significant research effort was directed by universities, NASA- and DOD-sponsored laboratories, and the analog and hybrid computer manufacturers to exploit the potential advantages of hybrid computation. This effort was directed at three basic problem areas: (1) the automation of hybrid computers, (2) the use of hybrid computation in new application areas, and (3) the development of new problem-solving procedures exploiting the advantages of hybrid computation.

With the redirection of the NASA and DOD programs in the late 1960s, much of the impetus for continuing work on these three problem areas was removed. The work on development of new applications and new problem-solving procedures had not progressed to the point where continued work on

automation of hybrid computation could be economically justified. With the prospects for continued development of the tool appearing bleak, work on new applications tended to slow down also. The net effect has been a general recognition that, while hybrid computers will continue to serve as valuable special-purpose simulation tools, the prospects for development of a general-purpose hybrid computation field are poor. Hybrid computers are, however, actively manufactured by several domestic and a number of foreign firms, and continue to find application in the aerospace and process industries and in education (Fig. 1).

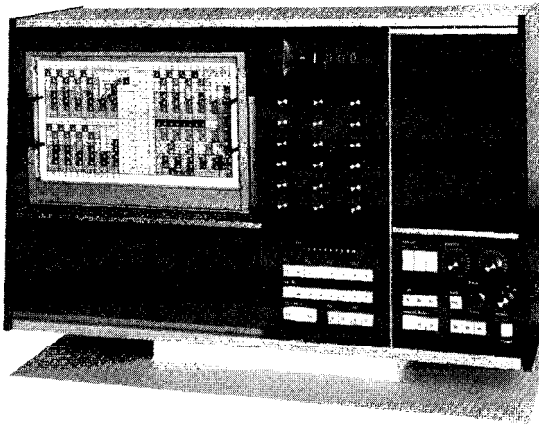


Fig. 1. EAI Mini-AC analog-hybrid computer system for educational and small engineering applications.

Despite this less than promising future for hybrid computation, there are several reasons why hybrid computation merits discussion and attention:

1. Hybrid computation had significant economic and intellectual impact on the development of computers and computational processes during a very active period.
2. Hybrid computers will continue to serve an important function in a number of application areas relating to high-performance dynamic system design and to the educational process.
3. Current predictions of the limited future growth of hybrid computation might well be wrong.

This article presents a brief overview of hybrid computer architecture and a method for classifying hybrid computation. Examples of successful hybrid computer applications are then discussed.

Hybrid Computers. As suggested in the preceding section, hybrid computers have involved many combinations of sizes of digital computers, analog computers, and interface systems. The interested reader is referred to Bekey and Karplus (1968) for a complete discussion. For purposes of the present discussion, a hybrid computer is a combination of a high-speed electronic analog computer and a modest size scientific digital computer linked together by a communication interface system. Fig. 2 presents a schematic organization of a typical medium-scale hybrid computer.

The analog computer is a collection of parallel computational elements that is hardware expandable; i.e., problem size capability is increased by adding more elements. Speed capability is increased by increasing the bandwidth of individual computational elements. These elements include summers, coefficient elements, integrators, multipliers, function generators, and logic elements. Once activated, the elements operate continuously and simultaneously in time until deactivated.

Precision is limited because all operations involve continuous signals (e.g., voltage as a function of time) of limited magnitude. Precision decreases with increasing signal frequency, but in modern analog computers it remains fairly high for signal frequencies in the kilohertz range. An analog computer program comprises a collection of elements interconnected to perform the operations necessary to solve algebraic and differential equations. In general, an analog computer is best suited for the solution of ordinary differential equations.

The digital computer is generally a serial computational device that is time-expandable; i.e., problem-size capability is increased by specifying a longer list of sequential instructions, which take a correspondingly longer time to be executed. All computation is performed in a central set of registers on numerical representations of variables. A digital computer program comprises the set of instructions specifying the order in which various arithmetic and logical operations are to be performed. Through proper programming, problem size and accuracy may be made arbitrarily large, but only at the expense of increased solution time.

The communication interface system is composed of three subsystems: control, data, and logic. The control subsystem affords the same degree of control to the digital computer program as enjoyed by a conventional analog computer operator; i.e., operational modes, time scales, coefficient element and relay settings, and state readout of all addressable analog and logic elements. The data subsystem

HYBRID COMPUTERS

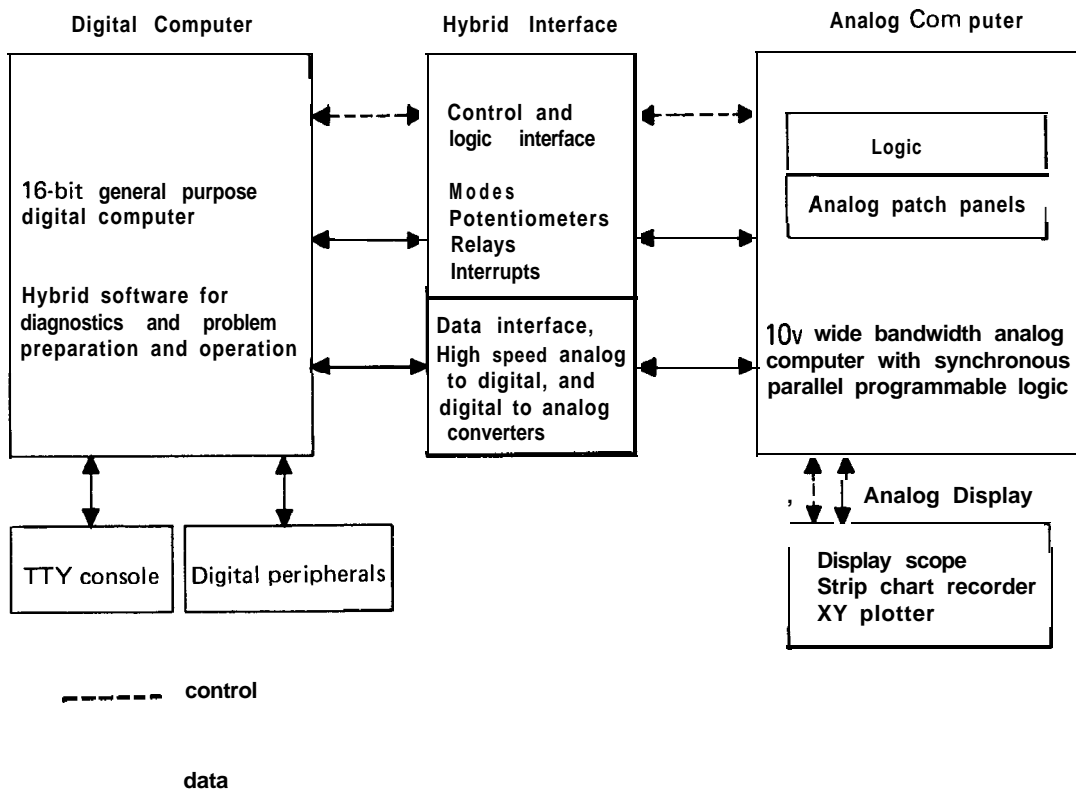


Fig. 2. Organization of a medium-scale hybrid computer. (Note: Dashed-line arrows indicate control; solid-line arrows indicate data.)

facilitates the high-speed communication of analog information between the two computers; it includes the high-speed analog-to-digital converter and multiplexer system and a number of digital-to-analog converters and multipliers. The logic subsystem consists of control lines that permit the digital computer program to set and reset logical conditions on the analog, sense lines that permit the digital to read logical conditions on the analog, and a set of interrupt lines that permit conditions on the analog to interrupt the digital program for immediate service.

By making optimal use of the best features of the two types of computers, one can hope to solve certain classes of dynamic system optimization and simulation problems several orders of magnitude faster than is possible with either analog or digital computers alone. The principal problem in hybrid computer programming is the effective utilization of the combined computer complex. The basic idea is to place the high-speed, low-precision, differential

equation integrations on the analog computer and the low-speed, high-precision computation and control on the digital computer.

The features of the hybrid computer that distinguish it from the digital and make it particularly well suited for a wide class of dynamic system simulation and optimization problems include:

1. **Speed.** For certain problems, a several hundred thousand dollar hybrid will demonstrate a speed that is orders of magnitude greater than that of a several million dollar digital computer.

2. **Man-Machine Interaction.** The user of a hybrid computer maintains an almost symbiotic relationship with the computer and its high-speed solution of his problem, through the extensive display and control features afforded by the analog computer.

3. **Analog Display.** The analog computer offers intuitively appealing high-speed (display scope and monitor scope) and low-speed hard-copy (stripchart

recorder and X-Y plotter) analog display capabilities.

Hybrid Computation. An analysis of different applications from many different fields of engineering and science indicates that the process of hybrid computation is best understood on the basis of control program structure rather than on the origin of the problem to be solved. This classification also contains certain useful information on hybrid computer requirements for satisfactory solution.

CLASSIFICATION OF APPLICATIONS. Hybrid computer programs can be classified into four basic levels on the basis of the structural features of the program. Different applications are then easily related to these levels.

Level 1. Automation of Standard Analog. Level 1 programs generally involve use of the digital to automate the setup, checkout, and operation of standard analog programs. All communication is through the control interface. A large majority of the work done on hybrid computers in installations with a long history of classical analog computation and many existing application programs fall under this classification.

Level 2. Digital Control of Analog. Level 2 programs generally involve use of the digital to control the high-speed operation of the analog computer; structurally, the programs are analog programs, with the digital computer interacting at the beginning and end of each analog solution. Most communication in Level 2 programs is through the high-speed data and logic interfaces, but, in general, it could be handled through the control interface.

Level 2 programs place little emphasis on digital computer speed and no emphasis on time synchronization of the two computers, since the computers alternate operation; i.e., the digital computes new starting conditions on the basis of previous analog runs, initiates a new analog run, waits for completion, etc. In effect, for Level 2 programs, the analog program appears as a high-speed subroutine to the digital program. Many automated design and optimization studies have Level 2 structure.

An example of a Level 2 program would be parameter optimization in a dynamic system where a model of the dynamic system is run on the analog computer and the digital computer program controls a search sequence varying parameters in the analog program, initiating a run, observing the results, modifying the parameters, etc.

Level 3. Function Storage and Playback. Level 3 programs generally involve use of the digital computer to store the results of one analog run for

playback to successive analog run(s). All run communication in Level 3 programs is through the high-speed data and logic interfaces. As with Level 2, Level 3 programs place little emphasis on the speed of the digital computer and the data interface, but Level 3 does require synchronization of the two computers.

An example of a Level 3 hybrid program would be a solution of a parabolic partial differential equation by a serial continuous space technique: (1) The analog integrates the resulting spatial differential equation at a specific time point, and (2) the digital provides sample information on the function from the preceding time point(s) and samples and stores the results of the analog calculations at the current time point.

Level 4. Parallel Analog/Digital. Level 4 programs are the common conception of hybrid programs; i.e., they involve the simultaneous (parallel) interdependent (hybrid) operation of both the analog and digital computers. Level 4 programs place great emphasis on both digital computer and interface speed, as the delay in these components acts as a pure phase shift to that portion of the problem proceeding on the analog. Level 4 programs can be difficult to program and debug because of the exacting synchronization requirements. An example of a Level 4 program would be where the digital program sampled one or more variables from the analog, computed some function of these variables, and sent the new function value to the analog program.

Level 4 programs are characteristically associated with large-scale simulations involving wide dynamic range requirements, extensive nonlinear function generation, and/or extensive simulation of digital subsystem components in a larger dynamic system simulation.

Each of the preceding levels can be further classified on the basis of whether the problem must be solved in "real time." Such a constraint occurs in any simulation involving real system components.

Fig. 3 presents characteristic activity versus time plots for each of the described levels. Also included in Fig. 3 are indications of the flow of information between the active periods of the digital program. This figure should clarify the described level classification of hybrid computation.

Conclusions. The future outlook for hybrid computation is not completely favorable. The problems of lack of generality, specialized programming, scaling, poor precision, and poor reproducibility inherent in analog programming tend to be very

HYBRID COMPUTERS

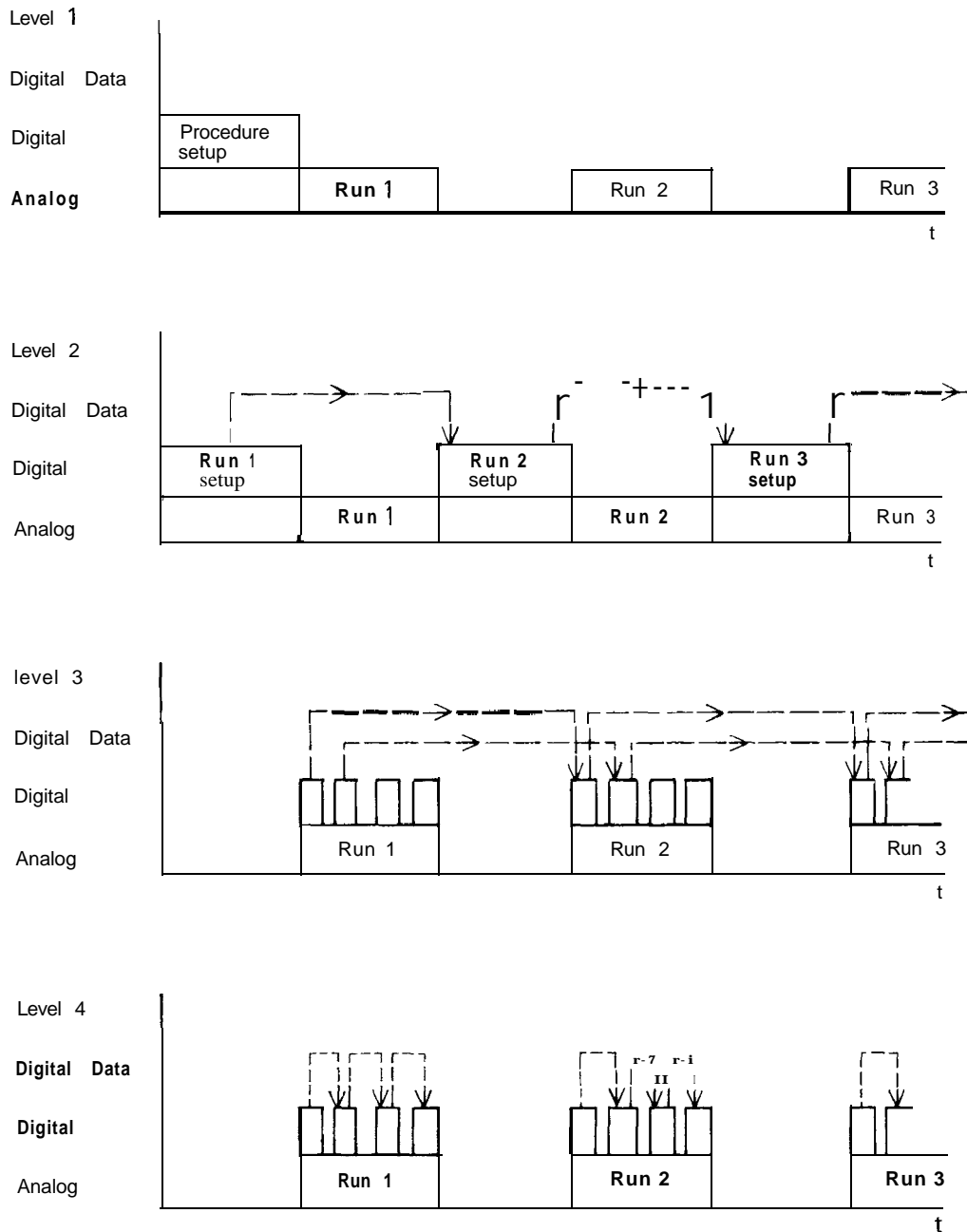


Fig. 3. Activity versus time plots for levels of hybrid computation. (Note: Levels 3 and 4 might also have the run setup activity of Level 2.)

difficult to overcome in hybrid programming. The great hope in combining an analog and a digital computer as a hybrid computer is to take advantage of the best features of both; it often turns out that such a combination accentuates the worst features of both in addition to introducing new problems of interconnection.

The best present estimate is that advances in digital simulation languages and techniques, parallel digital computer architecture, and interactive digital graphics will spell the long-term demise of hybrid computation in the form discussed here for all but very special applications. For the present, however, hybrid computers are extremely valuable in a wide

range of dynamic system design activities. Current research work on hybrid computation is contributing to a better understanding of the underlying mathematics and is paving the way for future development of computational systems for dynamic system simulation and optimization.

REFERENCE

1968. Bekey, G. A., and W. J. Karplus. *Hybrid Computation*. New York: John Wiley.

J. C. STRAUSS

IBI-ICC. See INTERGOVERNMENTAL BUREAU FOR INFORMATICS.

IBM. See MANUFACTURERS, COMPUTER.

IBM 360-370 SERIES

For articles on related subjects see **GENERATIONS, COMPUTER; MANUFACTURERS, COMPUTER;** and **OPERATING SYSTEMS:** Contemporary.

For articles on related terms see **CHANNEL;** and **VIRTUAL MEMORY.**

The IBM 360 and 370 systems are so widely used, and many features have been so widely imitated, that it becomes important for everyone in the computer field to be aware of aspects of their hardware and software systems. The literature concerning the 360 and 370 systems would fill a small library. The twentieth edition (IBM document GA 22-6822- 19, August 1972) of "IBM System/360 and System/370 Bibliography," which merely "identifies and describes all technical publications and related materials needed by those who plan, program, install

and operate the IBM System/360 (model 25 and above) and the **System/370,**" is itself a document that is 283 pages long. The structure of the central computer and its instructions are described in the manual (GA 22-7000) "IBM System/370 Principles of Operation," whose fourth edition, published in January 1973, contains 3 18 double-column pages. The reader who wishes to gain equivalent insight into the 360 and 370 software systems will not find any single concise manual, but can find a wealth of references in the above mentioned bibliography.

Many features of the 370 are the same as those of the 360, in some cases with minor variations. In the remainder of this article the reader can assume that statements about the 360 apply to the 370 as well. Specific statements about the 370 will not generally apply to the 360, except possibly to some atypical models like the 360 Model 67 and the 360 Model 195.

The basic unit of information in the 360 is the **8-bit** byte. Four bytes make up a word. Some of the instructions for the 360s operate on bytes, and others operate on half-words (two bytes), words, double words (eight bytes), and on strings of bytes. An instruction or operand address is always a byte address, the leftmost or most significant byte when a group consisting of more than one byte is being addressed. The **24-bit** address field permits the direct addressing of $2^4 = 16,777,216$ bytes.

The early 360 models were restricted in the size of central memory that could be used. The 360 Model **30** had a **maximum** of 65,536 bytes of central

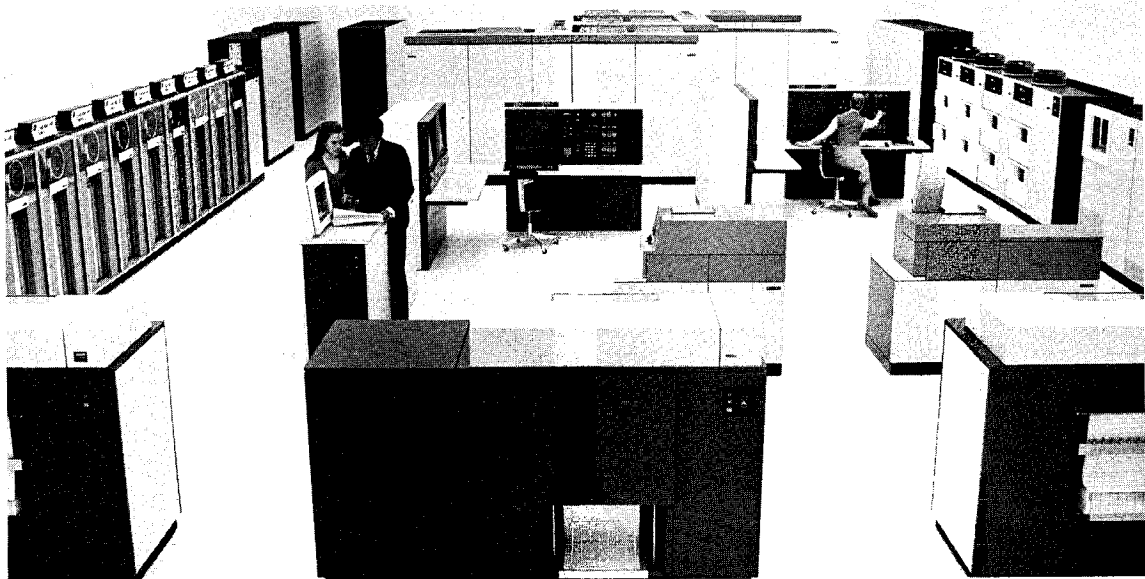


Fig. 1. The IBM 370 Model 168 multiprocessor system.

memory, and even the top of the line at that time, Model 75, could have only 1,048,576 bytes. The larger models could use auxiliary (though slower) large-core memory of up to 8,388,608 bytes. Large auxiliary, core memory has been eliminated in the 370 series, which permits even the small models to address a virtual memory that spans the full 16-megabyte address space. The amount of real storage permitted is still limited, though growing, on the smaller models. On the larger models, the original announcement limited the amount of real memory to 4 megabytes, but it was clear from the beginning that 370 systems with up to 16 megabytes of real storage would eventually become available. These very large memories are possible because the MOS large-scale integration technology used in the large 370 memories is much more economical than the earlier magnetic core technology.

A special version of the 360 Model 67, which dates back to 1966, provides for the use of full-word (32 bit) addressing, and permits the use of a $2^{32} = 4,294,967,296$ byte virtual memory. As of this writing it is not known whether this expanded addressing capability will be provided on any models of the 370 series.

The 360 design assumes that the computer will run under control of an interrupt-driven operating system. The system provides for automatic storage in main memory and for automatic loading from a different area of main memory of the contents of essential control registers in response to an interrupt.

The contents of these control registers may be considered to form a control word, which is referred to as the "program status word." The program status word contains the address of the next instruction, to permit resumption of a program after an interrupt. It also contains interrupt masks, the storage protection key, and a number of special control fields and control bits. One of these control bits distinguishes between system state and problem state.

The 360 has 16 general-purpose registers that serve as base registers and index registers, and which also serve as fixed-point accumulators and as temporary storage registers. A special instruction provided stores all or a subset of the general registers in memory starting at a specified address. Another instruction can load the general registers (all or a subset) from a specified area of memory. The general registers are 32 bits long, and the most significant 8 bits are ignored in address calculations (except on some of the Model 67 systems mentioned above). The use of the general registers as base and index registers permits the direct addressing of 16,777,216 bytes without requiring that each instruction contain a 24-bit address field. This is illustrated by the RX instruction format, one of several instruction formats used in the 360. Instructions may be one, two, or three half-words long. The RX format uses two half-words as follows:

Op Code	R1	X2	B2	D2
-----	---	---	---	-----

The 8-bit op-code specifies the instruction. There are two operands. The first is in the general register, specified by the 4-bit field R1. The second is in memory, at a location determined by adding the 12-bit displacement D2 to the contents of the two general registers specified by B2 and X2.

The 360 has a large and varied instruction set. There are three types of arithmetic: fixed point, floating point, and a special decimal arithmetic that uses strings of 4-bit binary-coded decimal digits as operands. There is a set of privileged instructions that can be executed only in supervisor state. Special instructions are designed to aid in the programming of data-processing applications, of which `TRANSLATE` AND `TEST` AND `EDIT` are interesting examples. A special `TEST` AND `SET` instruction helps in the handling of interlock problems.

The original 360 system permitted only one multiprocessor model, consisting of two Model 50 central processors. Multiprocessor models of several large 370 central processors have recently become available, with special instructions that permit the processors to signal to each other and to exchange status information.

The central processor has only very rudimentary input/output instructions to start and stop I/O, and to determine the status of an I/O operation that has been started or stopped. Input and output can proceed simultaneously, with computing under control of channels that can directly access main memory and which can execute "channel programs." Selector channels control devices like tapes and disks for fast high-volume data transfers. A multiplexer channel can control large numbers of lower-speed devices.

The most important new peripheral storage device introduced with the 360 series was the 23 14, a "direct access" disk storage system consisting of eight disk drives that use removable disk packs. A single controller permits reading or writing from only one disk pack at a time, but a number of seek operations may be proceeding simultaneously. Each pack can hold 28,000,000 bytes of information.

One of the most important new features of the 370 series is the 3330 series of disk systems that replace the 2314s and the block multiplexer channels that control these disk systems. The 3330 stores 100,000,000 bytes per disk pack (200,000,000 bytes in the Model II announced in 1973). The block multiplexer channel and a microprogrammed controller combine to permit much more efficient utilization of disk storage. Fig. 2 is a block diagram of a medium size 370 installation using 3330's for system residence and direct-access storage.

One of the objectives in the introduction of the 370 series was to permit even the small machines in the series to provide large amounts of disk storage, and also to provide communications-handling capabilities that were previously available only on relatively large and expensive models of the 360 series. The very small 370 Model 115, announced early in 1973, is perhaps less powerful as a processor than the 360 Model 30, but it has the full dynamic address translation capability and provides for direct attachment "without need for a separate control unit and selector channel" of up to four 3340 disk units. The 3340, which was introduced at the same time as the 115, has access characteristics similar to those of the 3330, but has a maximum storage capacity of about 70,000,000 bytes per pack.

The central processors of the small 370 models like the 115 and 125 actually consist of several microprogrammed processors in a single compact package. One of these processors executes the central processor instructions. A second handles the disk channel and control functions mentioned above. A third acts as a communications multiplexer, and permits a small number of terminals (up to eight on the 115) to be attached directly to the computer.

The original 360 concept assumed that only one major operating system would be required, which was given the name OS 360 (i.e., Operating System 360). It soon became apparent that many small and intermediate-size 360 systems needed a reasonably sophisticated operating system, but could not afford the high memory space and processor time overhead of OS 360. This led to the early development of an alternative system, DOS (disk operating system), which has been very widely used. Special operating systems were developed for the atypical 360 models such as the 44 and the 67. The time-sharing system, TSS 67, was a very major software effort. IBM also provided an alternative system, CP 67, which achieved greater acceptance than TSS among users of the Model 67. Still another alternative for the Model 67 is MTS (Michigan Terminal System) developed by the University of Michigan.

With the announcement of dynamic address translation as standard for the whole 370 line (except the already obsolescent 155 and 165, and the atypical 195), IBM introduced the VS (Virtual System) operating system. These new operating systems will presumably supersede the earlier OS and DOS systems. The dynamic address translation feature on the 370 series is quite similar to that provided on the 360 Model 67. A number of software systems written for the 360 Model 67 have been converted to

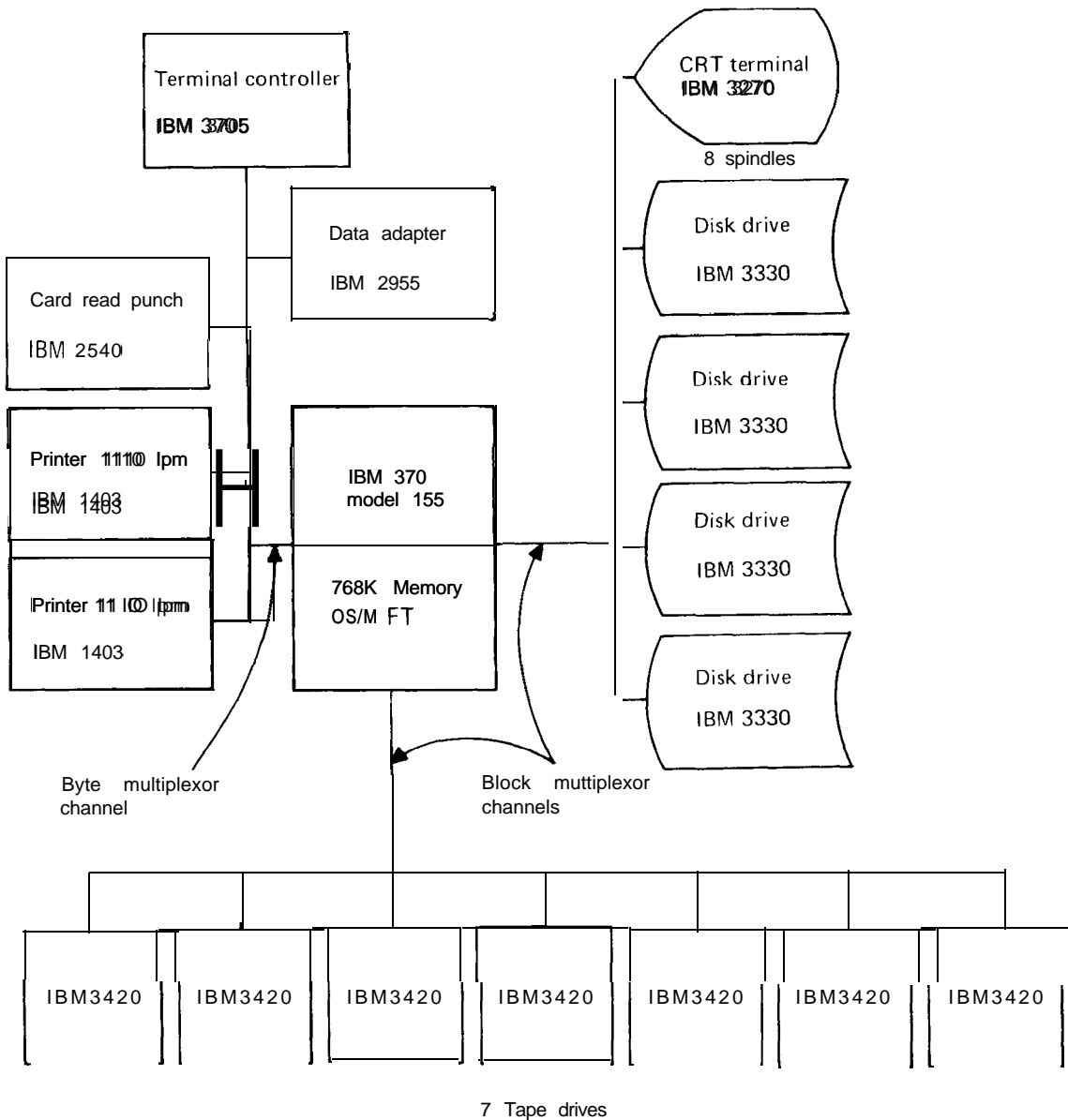


Fig. 2. Block diagram of a medium-size IBM 370 installation at Purdue University's administrative data processing center.

run on 370 systems. IBM's virtual machine system **VM/370** provides an extended CP67 system for the 370 series. The University of British Columbia has installed MTS on its **370/168**. The 370 version of MTS has been used at a number of other universities, including the University of Michigan.

S. ROSEN

IBM 1400 SERIES

For articles on related subjects see **DIGITAL COMPUTERS** : Early; **GENERATIONS, COMPUTER; MANUFACTURERS, COMPUTER;** and **RAMAC**.

The IBM 1400 series data processing systems were introduced to the business world in 1959. This line of equipment had a dramatic impact on the business data processing world at that time. The first of the 1400 series machines, the 1401, rapidly made obsolete the older vacuum tube and electro-mechanical "unit record" systems. The 1400 system enjoyed widespread use in every type of data processing application from 1959 until "third generation" equipment became available in the mid-1960s.

The 1400 line consisted of five basic computers: 1401, 1440, 1460, 1410, and 7010. The basic main frame, the 1401, was a second-generation, fully-transistorized machine with a magnetic core memory having original capacity options of 1.4K, 2K, and 4K characters, with later announced options of 8K, 12K, and 16K characters.

Internally data was represented in 6-bit BCD code, with additional parity check and "word mark" bits. The memory cycle was 11.5 μ s per character access.

Instruction formats were variable from one to eight characters, and data fields and records could be variable length within the constraints of the peripheral device characteristics and memory size. Instruction and data fields were defined by the presence of a word-mark bit set beneath the leftmost character of the instruction or data field.

The instruction format consisted of a single character op-code, two optional three-character data or instruction addresses, and an optional single character "d-modifier." The instruction set provided for internal data transfer, input/output control, add-to-storage decimal arithmetic, condition testing, and branching operations. A unique (at the time) single instruction provided for versatile printer-field

editing. Indirect addressing could be accomplished using any of three standard index registers.

The 1401 (Fig. 1) had an I/O interface that permitted only one I/O operation at a time to take place, regardless of the number of devices on line. I/O operations interlocked the central processor, although some overlap of processing and I/O operation could be gained by the addition of special features.

Although a wide variety of peripheral devices was available for the 1400-series, including MICR and optical readers, paper-tape readers, remote transmission devices, etc., the principal devices in use were:

1402 Card Reader/Punch. This unit, somewhat modified, is the presently used 2540. It is capable of reading 800 cards per minute (cpm) and punching 250 cpm.

1403 Chain Printer. This device had a maximum rated speed of 1100 alphanumeric lines per minute (lpm), was reliable and comparatively quiet, and had excellent print quality. The speed of this device and its relatively low cost was a significant factor in the widespread acceptance of the 1401.

729 Magnetic Tape Units. Seven-track units with speeds ranging from 15,000 to 62,000 characters per second (chps), depending upon the model and the recording density, which could be 200, 556, or 800 characters per inch (chpi).

7330 Magnetic Tape Units. These were relatively slow and inexpensive units (7,200 chps at 200 **chpi** density).

1405 Disk Storage Units. These were fixed-disk units with 50,000 or 100,000 directly addressable 200-character records. Records were accessed by a



Fig. 1. The IBM 1401 System.

IBM CARD

single arm moving laterally and vertically. Average access time was 600 ms.

1311 *Disk Storage Units*. These replaced the 1405-type units with modular disk-pack storage. Each pack stored 2 million characters in the form of 20,000 hundred-character records. The access device was of the "comb" type, requiring only lateral motion. Average access time per record was 250 ms.

Programming for the 1401 was accomplished through the use of several languages. It could be, and often was, programmed in machine language. A basic assembly language SPS (Symbolic Programming System) permitted the use of mnemonic operation codes, symbolic addresses, indirect addressing, data field establishment and definition, and was widely used. A significantly enhanced version of SPS, called "Autocoder"—analogous to basic assembly language for third-generation computers—became the predominantly used language. Autocoder used SPS constructs, but was free form (as opposed to fixed format for SPS statements) and employed macroinstructions for initiating I/O operations. Fortran and Cobol compilers existed, but were not widely used because of either excessive processing time needed for compilation or limitations on the scope of the language due to the limited memory size of the system. Various Report Program Generator (RPG) packages were available, as was a complete set of basic utility packages.

Operating systems were not used with the 1401, 1440, or 1460, although many users developed monitors to permit a rudimentary form of job control by automatic loading of a series of programs as opposed to the conventional method of loading each object program from a card reader before execution.

The 1440 system was initially a disk-oriented 1401 with slower peripherals and lower cost. Internally, the 1440 was, for all practical purposes, identical to the 1401 except that the memory cycle was 11.1 ms as compared to 11.5 for the 1401, and the printer and reader-punch buffers were relocatable in memory.

The 1460 was again basically a 1401 except that it had a 6 μ s memory expandable to a 32K capacity.

The 1410 systems, while having the same basic architecture as the 1401, were significantly more powerful. Memory sizes were 10K, 20K, 40K, 60K, or 80K characters. The internal code was BCD and the use of word-mark bits permitted variable length data and instruction fields. The memory cycle was 4.5 μ s per character. The instruction set was similar to the 1401, but included a table-lookup instruction and 64 different data-move instructions. Fifteen

index registers were available. The basic system had a single I/O channel, but a second could be installed. Like the 1401, the processor was interlocked during any I/O operation, though special features were available to provide limited overlap. Autocoder was the predominantly used language, although Cobol and Fortran were widely used. All peripheral equipment was the same as that used in other 1400 series systems except for 1301 and 1302 disk files, which were large fixed-disk units similar to the 350 units used on 305 "RAMAC" machines. The 1401 could be operated in emulated 1410 mode, which provided almost total compatibility with 1401 programs.

The 7010 system was functionally, although not architecturally, an advanced 1410. It used the 1410 instruction set and, unlike the 1410, had a 1410 compatibility feature. The 7010 accessed two characters in parallel on each 2.4 μ s cycle. Four I/O channels could be installed. Memory protection, an interval timer, and a program-level interrupt feature were available. All 1400 compatible I/O devices could be used on the 7010. Comparatively few of these systems were installed, as the system was introduced shortly before the System 360 was announced.

There were approximately 14,000 of the 1401 systems and over 1,000 of the 1410 systems installed. A typical 1401 system rented for \$8,000 per month and ranged from \$4,000 to \$12,000 per month. A typical 1410 system rented for \$11,000 per month and ranged from \$8,000 to \$18,000 per month.

The high-speed card reading and tape and printing ability of the 1401 systems ideally suited them for use as peripheral I/O systems to IBM 7000 series computers.

G. D. BAER

IBM CARD

For articles on related subjects see **MANUFACTURERS, COMPUTER**; and **NINETY COLUMN CARD**.

For articles on related terms and biographical information see **ASCII**; **EBCDIC**; **HOLLERITH, HERMAN**; and **WATSON, THOMAS J.**

In 1890, Dr. Herman Hollerith was faced with the problem of conducting the U.S. Census and evaluating all resulting information. The data from

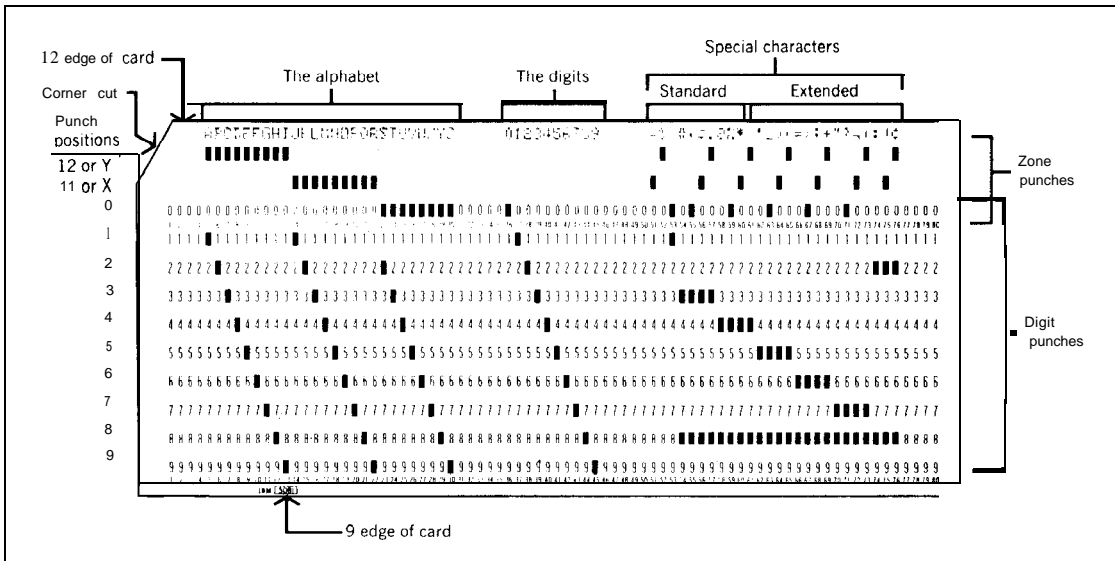


Fig. 1. IBM 80-column card format.

the previous census had taken ten years to process manually. To enable this processing to be carried out more efficiently, Dr. Hollerith invented a technique whereby information could be recorded in cards as a series of holes. He also invented a number of machines that were able to read these holes, interpret and accumulate this information, and sort cards into sequence. Using these cards and equipment, he was able to complete the 1390 census by 1892. This card, known almost universally today as the IBM card, is sometimes still called the "Hollerith" card.

Dr. Hollerith patented his ideas and later founded the Tabulating Machine Company. This company became the Computing-Tabulating-Recording Company in 1911. In 1914, Thomas J. Watson, Sr., joined this company, which later became International Business Machines Corporation.

While early cards enabled storage of an amount of information varying from around 30 characters to 90 characters, the card as it is known today has been standardized at a size of $7\frac{3}{8}$ in. by $3\frac{1}{4}$ in., and normally contains 80 characters of information. However, more than 80 characters may be contained in a card if special punching techniques such as binary punching are used.

The 80-Column Card and Card Codes.

Each card is divided into 80 columns and 12 rows (Fig. 1). Each column is normally used to record one character of information as a series of holes. The top row is called the "12" or "Y" row, and the second row from the top is called the "11" or "X" row. The remaining rows are called the 0, 1, . . . , 9 row, as

indicated by the digit printed on the card. Holes punched in the top three rows of the card are called "zone punches."

A decimal digit from 0 to 9 is readily recorded in a column by a punch in the appropriately valued row. In the case of alphabetic information, however, two punches are necessary in a column (see Table 1 for 80-column card codes). The first punch is a zone punch and the second punch is a numeric punch. Thus, the letters A to I are represented by a combination of a 12-zone and the numerics 1 to 9. Special characters (punctuation marks, etc.) are generally represented by a combination of one zone



Fig. 2. IBM 5496 data recorder used for punching 96-column cards.

IBM CARD

Table 1. IBM 80-Column and 96-Column Card Codes

Character	Description	80-Col. Card Code	96-Col. Card Code	Character	Description	80-Col. Card Code	96-Col. Card Code
b	Blank	No punches	No punches	E		12-5	BA 4 1
&	Ampersand	12	A8 2	F		12-6	BA 4 2
¢	Cent	12-2-8	BA 8 2	G		12-7	BA 4 2 1
.	Period	12-3-8	BA 8 2 1	H		12-8	BA 8
<	Less than	1 2 4 8	BA 8 4	I		12-9	BA8 1
(Left paren	12-5-8	BA 8 4 1	J		11-1	B 1
+	Plus	12-6-8	BA 8 4 2	K		11-2	B 2
	Logical "or"	12-7-8	BA 8 4 2 1	L		11-3	B 21
-	Minus	11	B	M		11-4	B 4
!	Exclamation	1 1-2-8	B 82	N		1 1-5	B 4 1
\$	Dollar	1 1-3-8	B 8 21	O		11-6	B 4 2
*	Asterisk	1 1 4 8	B 84	P		11-7	B 4 2 1
)	Right paren	1 1-5-8	B 84 1	Q		1 1-8	B 8
;	Semicolon	11-6-8	B 842	R		11-9	B8 1
┐	Logical "not"	1 1-7-8	B 8 4 2 1	S		0-2	A 2
/	Slash	0-1	A 1	T		0-3	A 21
#	Record mark	0-2-8	A8 2	U		0-4	A 4
,	Comma	0-3-8	A 8 2 1	V		0-5	A 4 1
%	Percent	0-4-8	A 8 4	W		0-6	A 4 2
_	Underscore	0-5-8	A 8 4 1	X		0-7	A 4 2 1
>	Greater than	0-6-8	A 8 4 2	Y		0-8	A 8
?	Question	0-7-8	A 8 4 2 1	Z		0-9	A 8 1
:	Colon	2-8	8 2	0		0	A
#	Number	3-8	8 21	1		1	1
@	At	" 8	84	2		2	2
'	Quote	5-8	84 1	3		3	21
=	Equals	6-8	84 2	4		4	4
"	Quotation	7-8	84 2 1	5		5	4 1
A		12-1	BA 1	6		6	4 2
B		12-2	BA 2	7		7	4 2 1
C		12-3	BA 21	8		8	8
D		12-4	BA 4	9		9	8 1

punch and two numeric punches, as shown in Table 1.

With the advent of computers in the 1950s, it was found necessary to extend the number of different characters that could be recorded on a card. Consequently, various combinations of holes are used to represent up to 256 different values, as in the Extended Binary Coded Decimal Interchange Code (EBCDIC), or up to 128 values, as in the American Standard Code for Information Interchange (ASCII).

The 96-Column IBM Card and Its Codes. In 1969, IBM introduced the System/3 computer and the 5496 data recorder (Fig. 2), which use a 96-column card rather than an 80-column card.

The 96-column card is physically smaller, measuring only $3\frac{1}{4}$ in. by $2\frac{1}{16}$ in. and can record 20% more data than the 80-column card (Fig. 3). Information is recorded on the 96-column cards in 32 columns, each column containing three sets of rows (Fig. 4). Each set of rows is called a "tier," and consists of 6 rows instead of the 12 rows found on the 80-column card. From the top down, the first row is the B row; the second, the A row; then the 8 row, the 4 row, the 2 row, and the 1 row. This series of rows is repeated again immediately underneath for the second set of 32 columns, and then again for the third set of 32 columns.

Numeric information is recorded as a combination of punches in the A, 8, 4, 2, and 1 rows, as shown in Fig. 4 and detailed in Table 1 for 96-

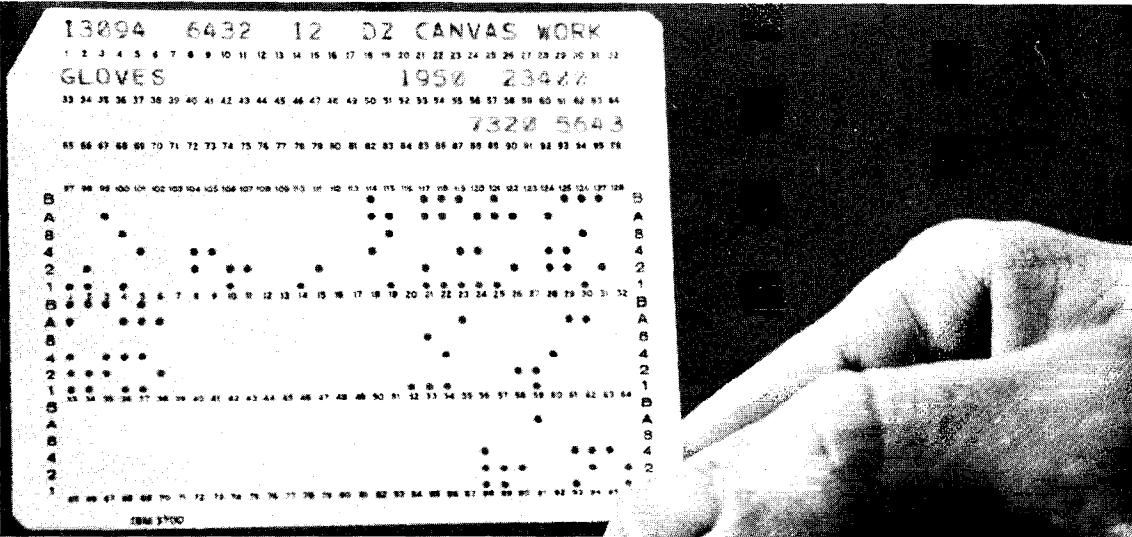


Fig. 3. The IBM 96-column card.

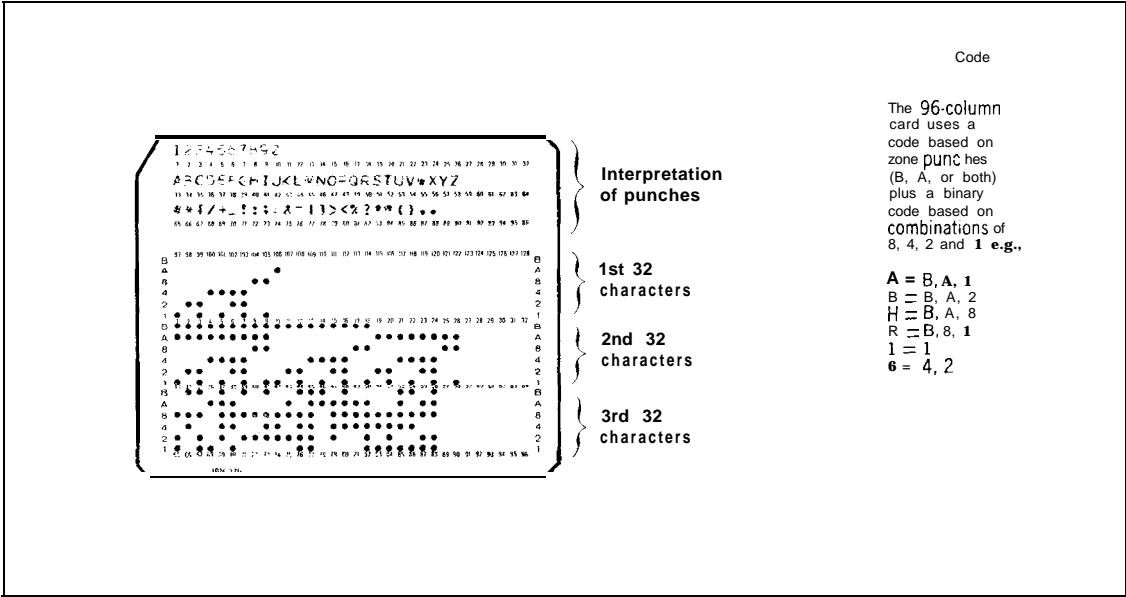


Fig. 4. IBM 96-column card format.

column card codes. Note that the B and A rows are used to represent zone punches, with both B and A corresponding to the Y zone punch on the 80-column card, B alone corresponding to the X zone punch, and A alone to the 0 zone punch. Note in Table 1 that the sum of the other punches on the 96-column card is equal to the second punch on the 80-column card for alphabetic characters.

Justification for the Use of Cards. The

main advantage of a card is that it is a separate unit record of information. Cards are used as a convenient means of recording information relating to transactions used to update master files. Thus, cards are useful for recording separate order transactions to update an orders file, issues and receipts transactions to update a product inventory file, or name and address changes to update a customer master file, for example. Cards are also used for the development of programs, with source program

statements being punched into cards for compilation and translation by a computer into actual computer instructions.

The use of computer terminals and time-sharing systems for program development since the late 1960s has resulted in significant improvements in efficiency of program writing and testing. These facilities enable the programmer to enter program instructions directly into the computer, using type-writer terminals, visual display terminals, or other hardware. It is probable, therefore, that the use of the card for programming will be reduced in the future.

The late 1960s introduced devices designed to replace the card as a storage medium for input of transactions into a computer. These devices and advanced techniques will undoubtedly become increasingly dominant, with the use of cards being reduced but not eliminated.

C. B. FINKELSTEIN

I CCP. See INSTITUTE FOR CERTIFICATION OF COMPUTER PROFESSIONALS.

IEEE-CS. See INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS-COMPUTER SOCIETY.

IFAC. See INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL.

IFIP. See INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING.

I M P . See INTERFACE MESSAGE PROCESSOR.

IOCS. See INPUT-OUTPUT CONTROL SYSTEM.

I PL-V. See LIST-PROCESSING LANGUAGES.

IDENTIFIER

For articles on related subjects see **PROCEDURE-ORIENTED LANGUAGES**; and **PROGRAMMING LANGUAGES**.

For articles on related terms see **LABEL**; and **PROCEDURE**.

In a programming language, an identifier is a string of characters used to identify (or name) some element of the program. This element may be a statement label, a procedure or function, a data element (such as a scalar variable or an array) or the program itself.

Most commonly, the word "identifier" is used almost synonymously with "variable name." In a system where the location of a program's data remains fixed throughout program execution, an identifier for a scalar variable is related to a memory address, which in turn references a physical location within the memory of the machine, which in turn contains a value. The intermediate relationships between the identifier and a value are usually transparent to a programmer, and thus some confusion arises in practice between the *name* of a variable (i.e., its identifier) and its value, which is the current contents of the memory location assigned to that identifier.

In the majority of programming languages, identifiers may be formed from any alphanumeric string of some restricted length (usually six to eight characters), provided the leftmost character is alphabetic. Some languages also permit the use of special characters.

J. A. N. LEE

IMAGE AND PICTURE PROCESSING

For articles on related subjects see **ARTIFICIAL INTELLIGENCE**; **CELLULAR AUTOMATA**; **CODES**; and **COMPUTER GRAPHICS**.

For articles on related terms see **AUTOMATA THEORY**; **GRAMMAR**; **GRAPH THEORY**; and **SOFTWARE**.

A wide variety of techniques exist for processing pictorial information by computer; these techniques are collectively referred to as *image processing* or *picture processing*. The information to be processed is usually input to the computer by sampling and analog-to-digital conversion of video signals obtained from some type of two-dimensional scanning device (television camera, facsimile scanner, etc.). Thus, at least initially, this information is in the form of a large array (e.g., in the case of ordinary television, about 500 by 500), in which each element is a number representing the brightness (and perhaps color) of a small region in the scanned image. The key distinction between image processing and *computer graphics* is that the latter does not deal with input pictures in array form, though it may construct pictures from input sets of coordinate data.

A digitized image array is sometimes called a "digital picture"; its elements are called "points," "picture elements," "pels," or "pixels." The values of these elements are typically six-bit or eight-bit integers. They usually represent brightness (or gray level); color can also be represented, but we will not deal with color here.

Picture Compression. A digital picture

may contain millions of bits, but most of the classes of pictures encountered in practice are *redundant* (in the sense of information theory), and can be *compressed* without loss of information. One can take advantage of picture redundancy by using efficient encoding techniques in which frequently occurring gray levels, or blocks of gray levels, are represented by short codes and infrequent ones by longer codes. If the pixels are encoded in a fixed succession (e.g., as in a television raster), one can capitalize on the dependency of each gray level on the preceding ones by encoding differences between successive levels rather than the levels themselves. If the dependency is very great, it may even be economical to represent the picture by the positions (or lengths) of runs of constant gray level, or more generally, to specify the positions and shapes of regions of constant gray level.

Except for very simple classes of pictures, only a limited degree of compression can be achieved using efficient encoding. However, for many purposes one need not insist that there be no loss of information; rather, one can approximate the given picture by another picture having lower information content. The digitization process itself, based on spatial sampling and gray-level quantization of a given real

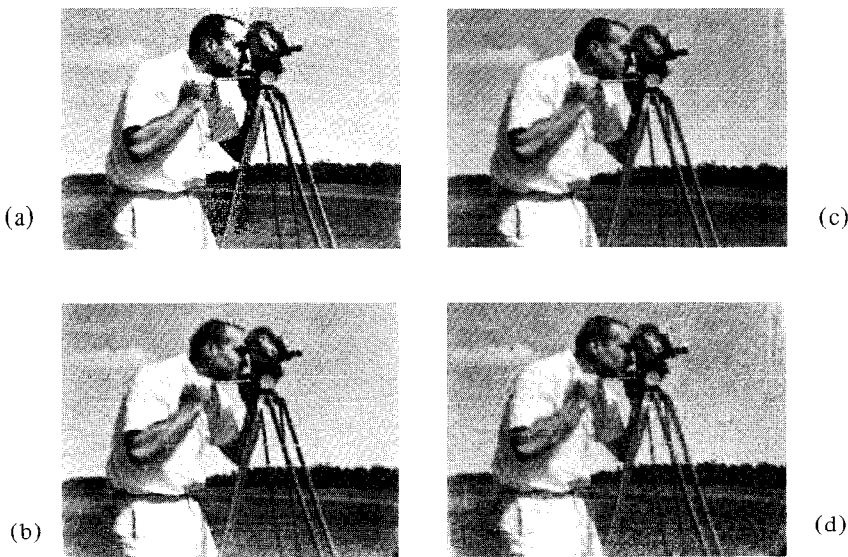


Fig. 1. Picture approximation. The original was a 256-by-256-point array of 8-bit values. It was divided into 256 16-by-16 point subarrays, and each of these was expanded in a two-dimensional Fourier series. In (a), the picture was reconstructed using only the first 128 coefficients of each series; in (b), using only the first 64 coefficients. In (c), the 128 coefficients of (a) were quantized to 4 bits, and in (d), to 2 bits, before reconstructing the picture. The average number of bits per point in these four approximations is (a)4, (b)2, (c)2, (d)1. (From P. A. Wintz, "Transform Picture Coding," *Proc. IEEE*, 60, July 1972, pp. 809-820.)

IMAGE AND PICTURE PROCESSING

image, is a process of approximation. In designing *approximation* schemes (Fig. 1), one can take advantage of the limitations of the human visual system; e.g., quantization can be coarse in the vicinity of abrupt changes in gray level. It is sometimes advantageous to apply approximation techniques to a transform of the given picture rather than the picture itself.

Image Enhancement. A picture is not always a satisfactory representation of the original object or scene; e.g., it may have a poorly chosen gray scale (under- or overexposure); it may be geometrically distorted; or it may be blurred or "noisy." Even if the nature of the process that degraded the picture is known, it may not be mathematically possible to invert the process. There are, however, many cases in which one can reduce the difference between a picture and its original by operating on the picture; this is the goal of *image restoration*. More generally, one can operate on a picture to improve its "quality" by making it more "contrasty," less blurred, or less noisy; this is the goal of *image enhancement* (Fig. 2).

One can "sharpen" a picture (increase local contrasts in it, deblur it) by emphasizing its high spatial frequencies. This can be done by multiplying the Fourier transform of the picture by a weighting

function whose values increase with distance from the origin, and then taking the inverse Fourier transform to obtain the sharpened picture. Similar effects can be obtained by performing a differencing operation on the picture (e.g., a "Laplacian") and combining the results with the original picture. (Doing just the differencing operation will convert solid regions into outlines). If the picture is not only blurred but also noisy, these methods may make it still noisier. It is often possible to achieve a useful compromise by emphasizing only a selected band of spatial frequencies (or analogously, using a **differencing** operation based on differences between average gray levels rather than between the gray levels of single pixels).

Similarly, one can "smooth" a picture by **de-emphasizing** its high spatial frequencies or simply by locally averaging it, but this is usually undesirable because it blurs the picture. Here, too, one can compromise by deemphasizing a selected frequency band; this is particularly effective when the noise results from the operation of a sampling process that is periodic (e.g., TV raster lines) or which at least has a characteristic "grain size" (e.g., photographic grain). If the noise is random, and several copies of the picture are available in which the samples of the noise are independent, smoothing without blurring can be achieved by averaging the copies. "Salt and

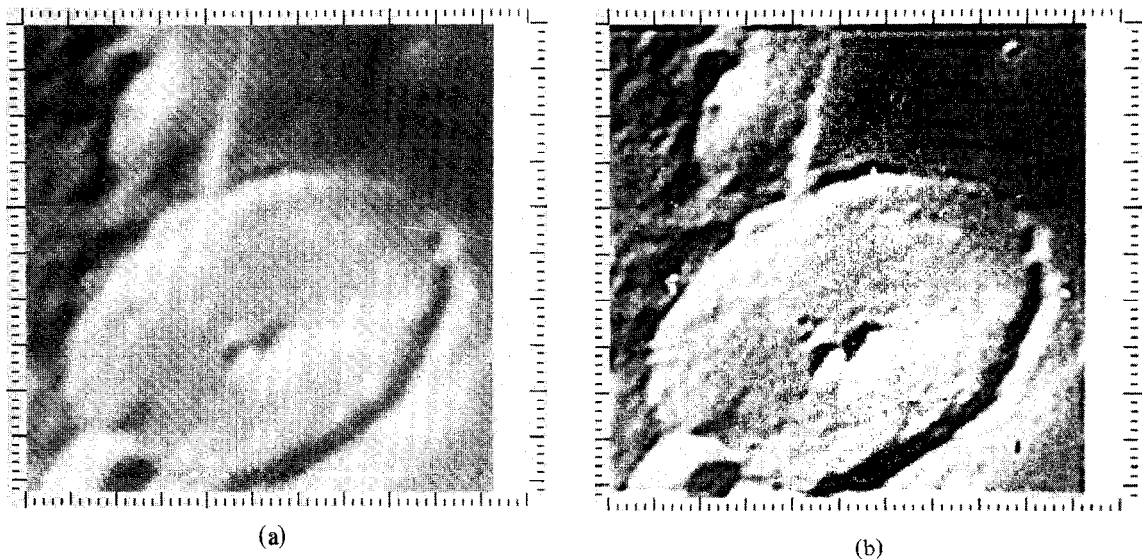
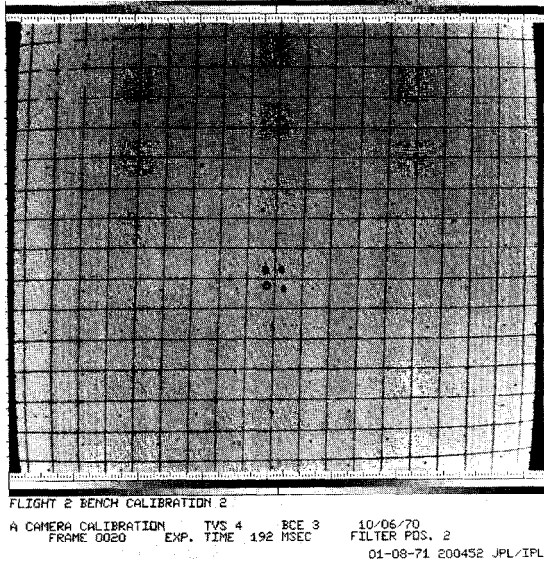
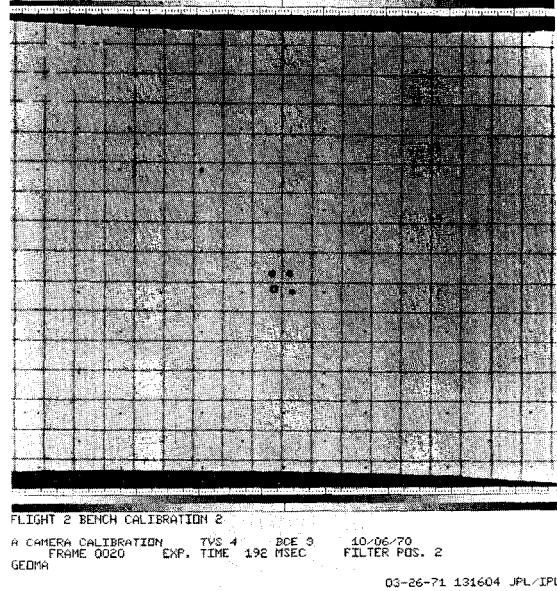


Fig. 2. (a) The lunar crater Gasendi, image blurred by atmospheric turbulence. (b) Results of enhancement by filtering to emphasize a high spatial frequency band. (From D. A. O'Handley and W. B. Green, "Recent Developments in Digital Image Processing at the Image Processing Laboratory at the Jet Propulsion Laboratory," *Proc. IEEE*, 60, July 1972, pp. 821-828.)



(a)



(b)

Fig. 3. (a) Mariner 9-grid target, showing geometrical distortion. (b) Results of distortion removal. (From P. A. Wintz, "Transform Picture Coding," *Proc. IEEE*, 60, July 1972.)

pepper" noise (e.g., TV "snow") can be reduced by performing a local averaging operation only at pixels where there are isolated anomalies in gray level.

A known geometrical distortion in a picture can be corrected by resampling it at an irregularly spaced array of positions (as specified by the distortion function) and outputting the samples as a regular array. Gray levels can be assigned to the new samples by interpolation from the levels of the nearby pixels. To correct an unknown relative distortion between two copies of a picture, one can find matches between pairs of distinctive local patterns (Fig. 3), measure the relative displacement of each pair, and construct a geometrical distortion function by interpolation from these displacements. (Local pattern matching is also used to extract relief information from stereopairs of pictures.)

Pictorial Pattern Classification. In picture compression and image enhancement, pictures are not only the input but also the output, since the goal is an approximation to, or an improved version of, the input picture. Another major branch of picture processing deals with picture *classification and description*; here the goal is the assignment of the picture to a category or, more generally, the creation of a data structure that contains useful information about the picture.

Pictorial pattern classification systems have been developed for many different applications, including optical character reading, analysis of nuclear bubble chamber pictures, medical diagnosis from micrographs and radiographs, chromosome analysis, recognition of faces or fingerprints, and interpretation of aerial photographs and satellite TV pictures. In such systems, the given picture may first be "preprocessed" to simplify or enhance it; a set of measurements is then made on it ("feature extraction"), and it is then classified on the basis of these measurements.

A wide variety of types of measurements or properties have been used for pictorial pattern classification. Important classes of examples include template properties (degree of match between the picture and a reference pattern, or template) and statistical properties (statistics of the gray levels in the picture (as in Fig. 4) or in a preprocessed version of it; these can be regarded as textural properties of the picture). Local properties, whose values depend only on small parts of the picture, are of particular interest because of their computational simplicity. If the classes of pictures are invariant under certain types of picture transformations (e.g., translation, rotation, change in contrast), one should use properties that are also invariant. One way of insuring this is to "normalize" the picture so that patterns

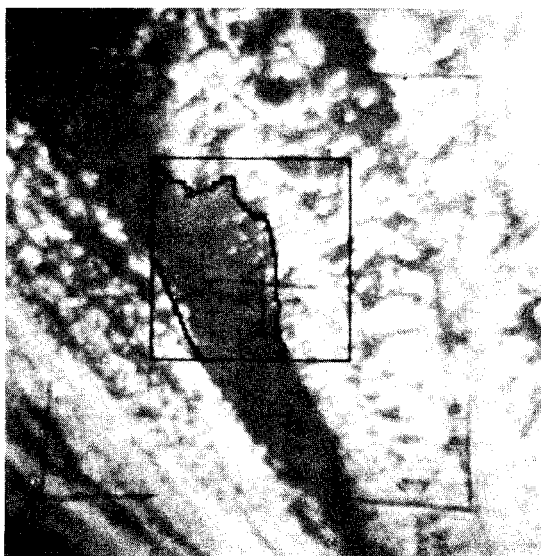


Fig. 4. Region outlining on a textured picture (a TIROS VI cloud-cover picture) using differencing of average gray levels, (From J. P. Strong, III, and A. Rosenfeld, "A Region Coloring Technique for Scene Analysis," *Comm. ACM*, 16, 1973.)

differing only by a transformation become identical before measuring the properties,

Picture Segmentation and Scene Analysis. Complex pictures, containing many different types of regions or objects, usually cannot be classified or described in a useful way on the basis of measurements made over the entire picture. Such a picture must first be segmented into parts (corresponding to regions or objects); once this is done, the picture can be described in terms of the parts, their properties (textures, sizes, shapes, etc.), and their spatial relationships. Picture segmentation and description are the goals of scene *analysis*.

Useful picture segmentations can often be achieved by preprocessing and then thresholding. For example, performing a differencing operation and keeping only the points where the difference value is high will tend to produce outlines of simple regions or objects; differencing of averages, rather than of single-pixel gray levels, can similarly be used to outline textured regions. Thresholding alone can be used to extract regions that are brighter (or darker) than their surrounds. It is usually advantageous to *track* line-like objects, or region outlines, rather than attempt to extract them all at one time; one can then adjust the extraction criteria from point

to point instead of applying a single criterion at all points.

Once a picture has been segmented into parts, new segments can be derived by extending, combining, or further splitting the parts. For example, given a picture segment, one can obtain its connected components, its border, its "skeleton"; or one can break it into parts on the basis of shape criteria. One can also measure geometrical properties of the segment (connectivity, area, perimeter, diameter, etc., as well as various shape properties). In addition, one can measure nongeometrical properties (e.g., textural properties) over the segment alone. The properties can then be used to construct descriptions of the picture.

Picture-Processing Software. There are as yet no programming languages specifically designed for picture processing, but a number of large software packages exist. Two of these that deserve special mention are Vicar (developed at California Institute of Technology's Jet Propulsion Laboratory) and Pax (developed at the Universities of Illinois and Maryland; originally a simulator for part of the ILLIAC III computer).

Vicar stores pictures as either real or integer arrays; in the latter case, the values of one or more pixels can be stored in a single machine word. This format permits fast execution of arithmetic operations on pictures using hardware instructions; it is thus very appropriate for image compression and enhancement work, which usually involves many such operations. The use of real arrays is important if the processing being done involves Fourier transforms or the like; if one truncates the values in a transform to integers, useful information may be lost.

Pax stores pictures as stacks of "bit planes"; the *i*th bit plane in such a stack is a binary array consisting of the *i*th bits of the pixel gray levels. One advantage of this format is that any number of "overlays," representing segments extracted from a picture, can be stored "in registration" with the picture by adding planes to the stack. Execution of logical operations on the binary planes is fast, since hardware instructions can usually be used to perform such operations on all bits of a machine word—hence, on many pixels—simultaneously. Thus, Pax is an appropriate system for scene analysis work involving many operations on picture segments.

Picture processing can often be greatly facilitated by using special hardware array processors. A variety of analog devices for processing images have

also been proposed. In particular, many useful picture-processing operations can be performed optically, but a discussion of nondigital processing techniques is beyond the scope of this article.

Picture Grammars and Automata. A variety of formalisms for computations on pictures and for description of picture structure have been studied. Formal grammars can be generalized from strings to arrays, and one can also define automata that have arrays as "tapes." Models for parallel computation on arrays—e.g., by "cellular arrays" of automata—are of special interest. "Perceptrons" (machines that compute linear threshold functions of local properties) have been extensively investigated.

Formalisms for computations on data structures (in particular, on picture descriptions) have also been developed. Grammars can be generalized from strings to graphlike structures, and automata having graph-structured "tapes" can be defined.

One can regard picture segmentation and scene analysis as parsing operations with respect to such formal models. However, a purely "syntactic" approach to scene analysis is unlikely to be adequate except in simple cases. In general, it seems necessary to develop "knowledge-based" scene analysis systems, which make use of "semantic" information about the objects whose images appear in the scene. This approach is being actively pursued in connection with the development of robot vision systems; it constitutes an important area of artificial intelligence research.

REFERENCES

1969. Rosenfeld, A. *Picture Processing by Computer*. New York: Academic Press. (A highly condensed version is in *Computing Surveys*, Vol. 1 (September 1969), pp. 147–176.)
1970. Andrews, H. C. (with contributions by W. K. Pratt and K. Caspari). *Computer Techniques in Image Processing*. New York: Academic Press.
1972. *Proceedings Of the IEEE*. Special issues on digital image processing and digital pattern recognition (July and October).

A. ROSENFELD

INCREMENTAL COMPILER. See COMPILER, INCREMENTAL.

INDEX REGISTER

For articles on related subjects see **ADDRESS MODIFICATION**; **ADDRESSING**; **GENERAL REGISTER**; **INDIRECT ADDRESS**; **MACHINE INSTRUCTION SET**; and **REGISTER**.

For article on related term see **OPERAND**.

An index register is a storage device most often used in the determination of an operand address, but which may be used for other purposes, mainly as counters.

In the process of the formation of an address of an operand, one can distinguish three basic parts. Consider, for example, the **ADD** instruction in a program loop computing the sum of the elements of a vector. The operand address of the **ADD** instruction is formed from:

1. The address of the base of the vector (its first element) relative to the program module. This address is known when the program is being written.
2. The memory address into which the program module is loaded. This address is known at load time.
3. The offset from the base of the vector, which depends on the element that is currently being added and which is known only at execution time.

Index registers are normally involved with the last of the three parts of the address.

The address computed within the index register is referred to as the effective *address*. The index register accomplishes its role of forming the effective address in one of two ways: Either the address is formed from a constant in the address field of an instruction plus a changing offset in the index register, or the address as a whole is contained in the index register. In the former case, shown in Fig. 1, the index register is used as a counter.

The number of index registers in a machine and the number of index registers used in the formation of the effective address and other attributes of the index registers are highly dependent on the particular architecture. Thus, one finds machines with a single index register, one index register and one dedicated base register, multiple index registers and/or base registers, and machines in which the general-purpose registers may be utilized for indexing and base addressing. The aforementioned possibilities by no means cover all varieties. The number of index registers utilized in forming the effective address is usually one or two; i.e., the address is

INDIRECT ADDRESS

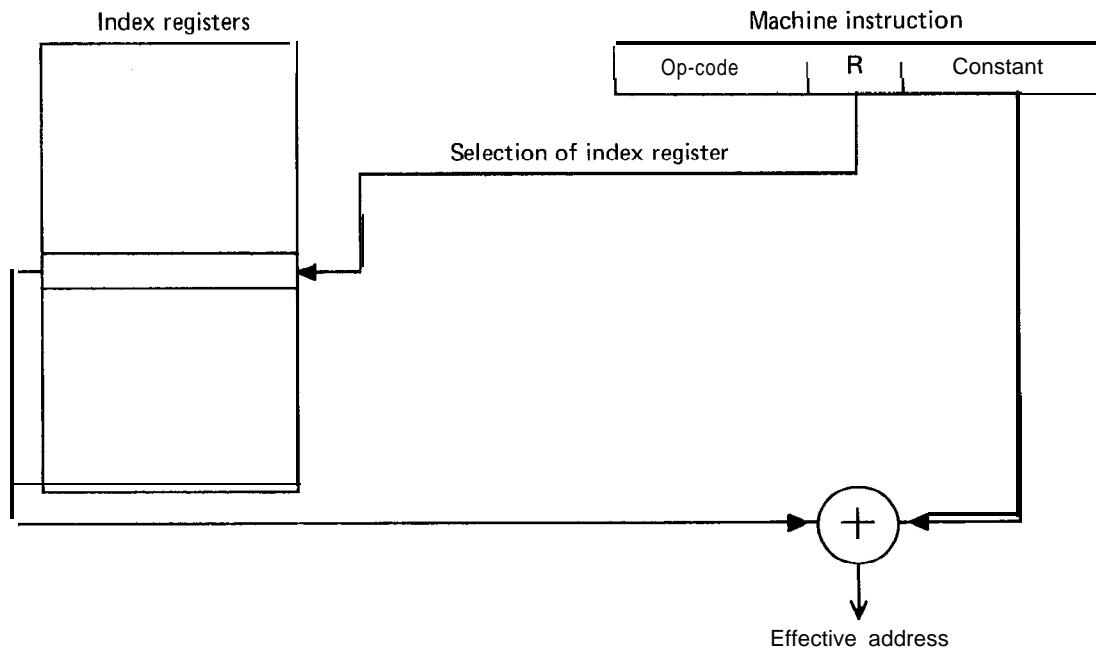


Fig. 1. Example of the formation of an effective address.

formed from the constant part of the address field plus the contents of one or two index registers.

As mentioned before, apart from their function in address formation, the index registers can serve as counters. As such, they are used in special instructions that increment or decrement the contents of the index register and check its new contents, thereby exercising control over the program flow.

Special care must be exercised in the use of index registers when the computer possesses an indirect addressing mode. In this case, the index register can be used either to compute the location of the indirect address (pre-indexing) or as an offset to the indirect address itself (post-indexing). When more than one index register is involved in the formation of the effective address, both pre- and post-indexing may be present. Again, the availability of either of the modes varies widely among different machines.

G. FRIEDER

INDIRECT ADDRESS

For articles on related subjects see **ADDRESSING**; and **MACHINE INSTRUCTION SET**.

For articles on related terms see **OPERATION CODE**; and **VIRTUAL MEMORY**.

A simple computer instruction contains an operation code and an address that points to a location in memory. The contents of that location may be the data required by the operation, or may be an address that points to another location in memory. In this latter case, the address in the instruction itself is called an "indirect address," since it references data indirectly by pointing to the address of the data rather than to the data itself.

In some computers, the instruction itself contains a control field (one bit per address is enough) that specifies that the corresponding address is an indirect address. In other computers, tag bits are associated with data words, and these tag bits determine whether the word is to be treated as data or is to be used as an address that points to data.

Many systems support multilevel indirect addressing (see Fig. 1). The address retrieved in the memory word may itself be an indirect address that points to another memory location, which in turn may be an indirect address, etc. Computers that allow multilevel indirect addressing usually have a time-out interrupt facility that causes an interrupt to occur in the case of a nonterminating indirect addressing loop.

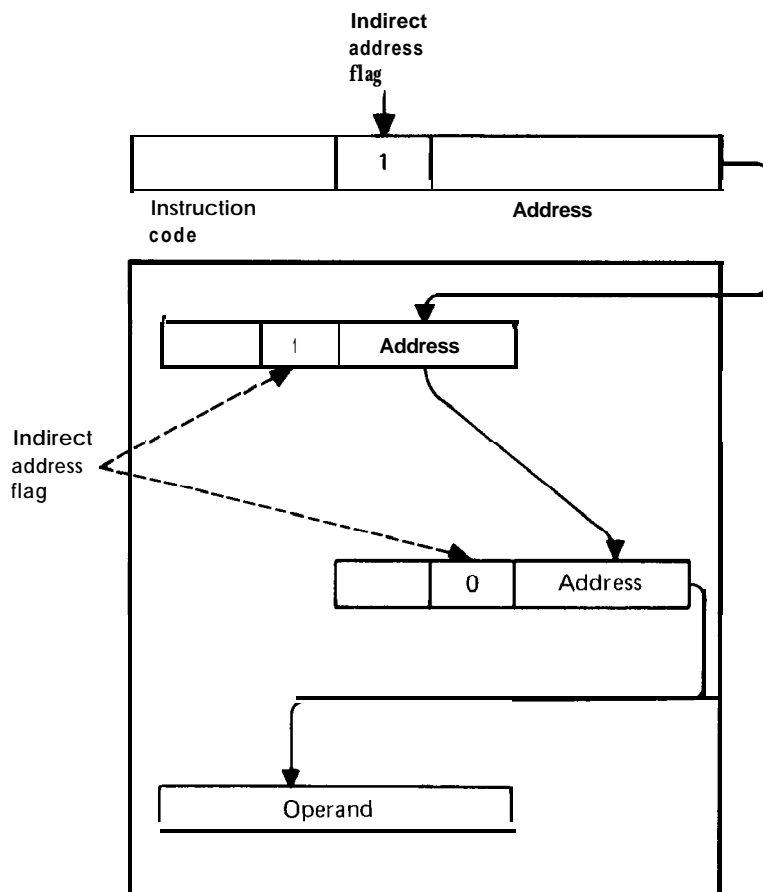


Fig. 1. Two-level indirect addressing.

There are many uses for indirect addressing. It has been used most effectively in those systems that require a longer address field than can be conveniently or reasonably provided in each instruction. Many small computers use indirect addressing in this way, but it is also used on many larger ones. Thus, the “descriptors” on large Burroughs systems are indirect addresses in which the address word contains the origin and size of an array that is addressed, to permit an automatic check for out-of-bounds addressing. The Multics system uses two-word indirect addresses to permit the addressing of its very large virtual memory.

S. ROSEN

INFORMATION AND DATA

For articles on related subjects see DATA STRUCTURES; DATA TYPE; INFORMATION RETRIEVAL; and SYMBOL MANIPULATION.

For article on related subject see SYNTAX, SEMANTICS AND PRAGMATICS.

Data processing is used to produce data that provides people with information to support their decisions or actions.

Information may be defined as knowledge, especially as it provides people (or machines) with *new* facts about the real world. *Data* may be defined as physical symbols used to represent information for storage, communication, or processing. To determine what data to use, it is important to decide what information the data is to represent. Ignorance of this fact has caused much trouble in data systems.

INFORMATICS. See INFORMATION SCIENCE.

INFORMATION AND DATA

Statistical information theory (or the theory of signal transmissions) uses a special and very limited concept of information which is more related to data or signals than to knowledge or meaning. Thus, this theory has been useful in certain design problems regarding computers or communication technology. However, it does not contribute to the design of information systems or data processing systems and will not be discussed further in this article, where we are interested in information as related to data and to real-world concepts.

Information Systems. To be supplied with useful information to improve our knowledge, we use information systems. The system information is represented by recorded or transmitted “physical” symbols, which we call “data.” To know which data to have in the data base of the system, the user needs to determine first what information he can use. To retrieve the right data, the user could be requested to specify what information he wants retrieved. He cannot, in general, be expected to specify structural characteristics of the data system, introduced for technological or computing reasons. In specifying information, a user selects terms that refer to details of his view of the real world.

Production of the desired data often requires a computation based on other data. To find which data is needed to produce some specified data, we have to determine what information we want to produce and what other information must be used for the production. We thus need to study *information precedence relations* before we can determine data needs and computation structure.

To determine which information and processing to have in the system, the cost of collecting, storing, and updating must be estimated and balanced against the estimated utility of all its known uses. Some of this must be done informally, but some can be done in a formalized way, using the information precedence relation structure (e.g., analyzing the associated information precedence matrix).

The information aspect (or “infological” aspect) of data imposes the need for the user to specify what information he desires so that the system can provide him with appropriate data incorporating this information. It is also important that his programs and queries will be unaffected (by ignoring the actual data structures implemented) by much of the restructuring required when technology or usage changes. This aspect of data is often also called the *relational aspect*, but in this case a somewhat more formalistic view is usually taken.

Information, Reality, and Data-Semiotics. In information systems we use symbols, which, through their association with reality, provide information. We take data to mean *physical* symbols that can be recorded or transmitted or processed. *Semiotics*, the theory of symbols, embraces data, information, and our understanding of reality. Semiotics is concerned with *syntactic*, *pragmatic*, and *semantic* aspects of symbols. **Syntactics** treats relations between the symbols themselves (including processing); **pragmatics** treats the relation of symbols to behavior; and **semantics** treats the relation between symbols and what they stand for or designate. Fig. 1 depicts this “semiotic structure”. “Symbol” is often used, and notably in semiotics, to refer to both physical and conceptual symbols without a clear distinction being made. However, we are interested in both applications and in their inter-relations.

We use the word “data” to refer to physical symbols and let “data term” (or “data item”) stand for the singular form. Then we use “conceptual terms” or “conceptual symbols,” which are reference concepts, as our basic concepts of reality, i.e., as the elements of the users’ *conceptual model of the real world* and their frame of reference.

For example : “Tom is a boy” is a string of written symbols or data. “T,” “O,” and “m” are three symbols that form the composite symbol “Tom,” which is a data term. The word “boy” is another

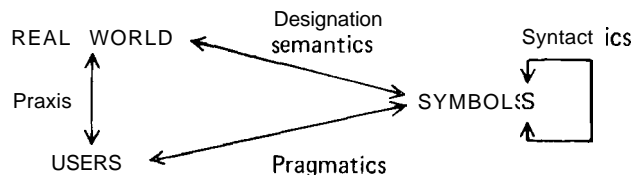


Fig. 1. Common semiotic structure that ignores the distinction between physical and conceptual symbols, and between conceptual symbols and reality.

data term, and the string "Tom is a boy" is made up of four data terms.

The data term "boy" is associated by me (as I read it) with the word "boy," which again I associate with the conception of a boy in general (or the idea of all boys). Thus, the word "boy," as perceived by me, acts as a conceptual symbol associated with the concept *boy*.

An important observation, fundamental to all formal treatment of reality, is that the physical symbols, or data, can refer only to our conceptual model of reality, not directly to reality itself. The conceptual terms are the bridge entities between reality and information and between information and data, thus also bridging data and users. The *designata* of data are the conceptual terms of the users. In Fig. 1 we have used the word "praxis" to name the relation between users and the real world. This is to indicate that it is the interaction between the users of the data, information, and the real world (i.e., what occurs in their practical experience) that is of importance. We are now led to a more complex diagram, Fig. 2, where information enters.

Extended Semiotic Structure. In Fig. 2, information is introduced into the diagram as the necessary intervening concept to explain how data is able to refer to the real world or is able to influence

the users (pragmatics). Note that "users" have to know or determine the information structure (model and conceptual symbols) to be able to specify the data they need, or to understand the data provided to them. Semantics embraces the relation of data to conceptual terms and model. The conceptions are formed in the minds of the users as a result of "praxis," their practical experience.

From the diagram in Fig. 2, we simplify the structure in Fig. 3 and consider the three aspects of the user's requirements: reality, information, and data.

ELEMENT OF REALITY: E-SITUATION. Information tells what is held to be true-in reality. In a sense, an element of information is a truth statement about an element of reality, and it can reveal, at most, our conception of that element of reality together with a simple property, state, or behavior of it at a certain interval of time. "Simple property" refers to one of all properties prevailing in our conception of that element or object. We may call this an "elementary situation" (an e-situation).

ELEMENT OF INFORMATION: E-MESSAGE, E-FACT. An element of information can be seen as a message that informs about an elementary situation. We will refer to such as an "e-message" (elementary message). It is seen that an e-message consists of

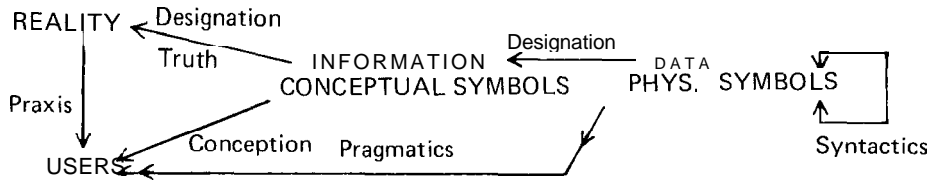


Fig. 2. Extended semiotic structure.

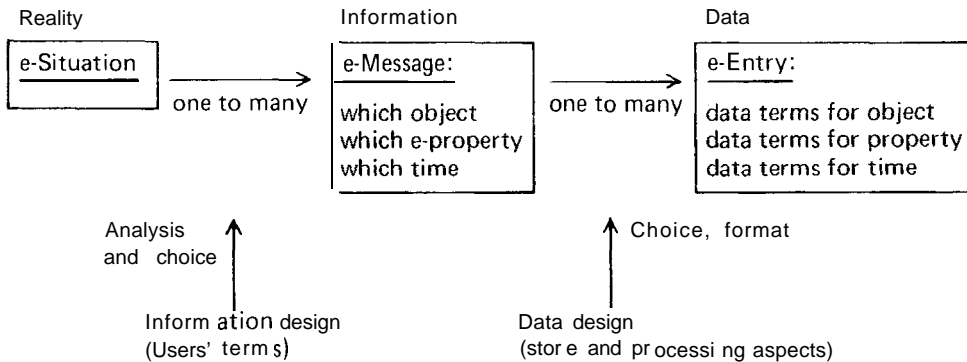


Fig. 3. User's requirements: reality, information, and data.

INFORMATION AND DATA

three conceptual terms (conceptual symbols):

e-message: $\langle 0, A = a, t \rangle$

where o : object (e.g., Article XY257)
 A = a : simple property or behavior
 (e.g., Manufacturer
 = COMPANY X)
 t : time (e.g., since 1970)

An e-message is a structurally minimal message in the specific sense that if one of its terms is deleted, the rest do not inform of an e-situation.

Formally, an e-message may be regarded as an instance of a *binary relation that is a function of time* (and thus is a ternary relation). The relation has values in the three domains associated with the conceptual terms. While the relational view is formally equivalent, and thus equivalent in data processing with the e-message, the two may be rather distinct from the user's perspective. An e-message is true or *false*, depending on whether or not a corresponding e-situation existed at the specified time. A true e-message is an *e-fact*.

In the literature, “associative triple” is often used as almost equivalent to e-message (disregarding the time dimension). The typical “associative triple” has the form, “*attribute of object is value*” (e.g., “profession of Tom is teacher”). It does not mention at what time interval it is true.

ELEMENT OF DATA: E-ENTRY OR E-RECORD. The link to the data representation of information is obtained through the design of one group of data terms for each of the conceptual terms in each e-message. The set of this data will be referred to as an "e-entry." If the data is stored in the same physical area, it may also be referred to as an "e-record". Thus, there is first a step of information analysis and design before the data design can start (whether we intermix these steps or not). Of course a written description of an e-message (like the preceding one) is already an e-entry, but data design will consider which data and formats (ultimately bit configurations) to use in the system for each e-message (see Fig. 3).

Kind of Information, Kind of e-Messages, e-files. In our conception of reality we use *kinds* or *classes* of objects and kinds or classes of properties. Thus, e-messages that inform about objects of the same kind by assigning properties of the same kind to them may be regarded as *instances* of the same information kind or elementary concepts, or “e-concepts” as we may call them. The kinds of

objects and properties conceived are subjective, and sometimes differ strongly among subjects. To achieve a satisfactory intersubjectivity in choosing terms is thus an important part of information design.

The classification of objects, properties, and e-concepts is fundamental to information system design. Without it, we could not organize data for retrieval. Also, algorithms must be designed for the e-concepts in order to hold for all e-messages of the same e-concept. An e-concept is associated with real-world e-situations of the same kind.

As an aid in resolving subjective differences and defining synonyms, *thesaurii* are designed, which define the conceptual terms used. Now we have the form

e-message: (oc; pc; tc | oi; pi; ti)

where *oc*, *pc*, *tc* = class names that denote the class of the object, property, and time indication, respectively.

oi, pi, ti = identifiers in the domain of the respective classes and give the value *tuple* of the message.

For example, the e-message

“article A ordered by customer C at date 730328”

could appear as

e-message: (article; ordered by; at date | A; C;
730328)

Each class may, of course, be defined hierarchically, in which case new class names are added and associated identifiers are added to the e-message value tuples. The class names are invariant for all e-messages of the same e-concept. They thus characterize the following concept:

e-concept: $\langle \text{oc}; \text{pc}; \text{tc} \rangle$

or

e-concept: (oc, pc)

when the class of time indication may not be of interest, as in early stages of analysis.

Thus, the e-concept has a simpler description than the e-message. It is therefore the natural thing to specify first in information design.

Example

Object class Property class

e-concept: <article; ordered by, customer>

Note that "customer" has the role of a component of the property class specification, although in the general frame of reference it appears as an object class. A collection of e-entries for e-messages of the same e-concept form an *e-file* (elementary file).

TRANSPOSITION (OR PERMUTATION) OF E-CONCEPTS-RELATIONS. In the preceding example, the e-concept was seen as information about articles while also informing about the kind of e-situations. Alternatively, information about this kind of e-situation can be about "customers" or about "ordered by":

e-concept' (customer; ordered, article)
e-concept'' (order; article, customer)

We say that we *transpose* an e-concept when we change the roles (as object part or property part of the e-message) of the conceptual terms. Note that transposition implies both a permutation of the terms *and* a change (e.g., a reversion) of the relation involved (e.g., precedence is changed to a **SUCCESSOR**). Transposition is not as so simple a change in the system as it may seem. Not only are terms permuted and changed, but the retrieval structure (e.g., sorting order) will probably be changed also. The terms in the e-concept descriptions are the basis for the *primary keys* to be used in the system.

In the same way that an e-message could formally be regarded as an instance of a relation, an e-concept is formally a time-dependent binary relation and thus a ternary relation. (When the conceptual terms are hierarchic groups, the relation appears as n-ary, $n > 2$). In such a case, one ignores to some extent the distinction between "object" and "property" (and maybe "time") and regards these as two domains with a relation. The possibility of transposition seems to support such a view. Still, it seems that there are many reasons to retain the distinctions, as in the infological view.

For instance, the binary relation

product group XY, amount sold, 10000\$

becomes the ternary relation

product group XY in district D, amount sold,
10000\$

where the object is product group XY in district D. It becomes a 4-relation if we add the time:

product group XY in district D, amount sold,
10000\$, during 1973.

E-ALGORITHM, E-PROCESS. An elementary algorithm, or e-algorithm, produces e-messages of one e-concept while using other e-concepts as precedents (or input). An execution of an e-algorithm is an e-process that produces an e-message. An e-algorithm is an implementation of the precedence relation between the e-concept produced and its precedent e-concepts. Common algorithms are systems of e-algorithms and e-concepts.

CONSOLIDATION OF E-CONCEPTS AND E-MESSAGES: C-CONCEPTS, N-RELATIONS. Often One is interested in several properties of an object at the same time, calling for several e-messages about the same object, as in Fig. 4(a). One may even be interested in properties of the situation itself, and thus will use e-messages about an e-message. This may be because they are naturally conceived (by some users) as belonging together.

In this case we have a *naturally consolidated* information kind [Fig. 4(b)] or *natural c-concept* (*c* for consolidated). It may then be a natural n-ary relation or n-relation (time dependent), *n* being less than or equal to the sum of orders of relations of the e-concepts involved.

Another reason for consolidation is that it will save data transport (e.g., access and transfers) and equipment (e.g., tape handlers). Then it is rather a question of data design and consolidation of e-files into c-files. This is not a natural n-relation. Consolidation of e-messages about the same object (and time) will save a number of occurrences of object (and time) references. Consolidation usually is combined with formatted records or entries so that a common record description can be used for a file and each entry is just the tuple of value terms.

Example. The message **m0**:

m0: article # 325; order; quantity 5 pieces;
customer # 127; date 720911;

may be seen as a consolidation of two e-messages:

e-m1: article # 325; order; customer # 127;
date 720911;
e-m2: e-m1 : quantity 5 pieces;

The e-message e-m1 informs about the object "article # 325" and the e-message e-m2 informs about

	Object	Property	Value	Time
e-message 1:	Prodgrp K, article #325,	order,	customer #1 27,	date 720911;
e-message 2:	e-message 1, quantity,	5 pieces;		
e-message 3:	Prodgrp K, article #1 7,	order,	customer #1 27,	date 720911;
e-message 4:	e-message 3, quantity,	11 pounds;		
e-message 5:	Prodgrp M, article #1 2,	order,	customer #1 27,	date 720911;
e-message 6:	e-message 5, quantity,	7 gallons;		

(a)

Order		
Date	720911	
Customer	#1 27	
	Product group	Article
	K	325
		5 pieces
		17
		11 pounds
	M	12
		7 gallons

(b)

Fig. 4. (a) Collection of associated e-messages. (b) consolidated message ($e\text{-}m1 \cup e\text{-}m2 \cup e\text{-}m3 \cup e\text{-}m4 \cup e\text{-}m5 \cup e\text{-}m6$).

the object "e-m 1"; that is, e-m2 informs that the situation described by e-m1 has the further property "quantity = 5 pieces," so that e-m1 and e-m2 together give the information of m0. Here "order" and "quantity 5 pieces" are regarded as two simple properties.

In a typical, simple query to an information system or a data base, one wants to know the value of a stored instance of a certain term. One cannot, of course, just ask for the term. For instance, one cannot just ask: "What is the quantity?" It is, on the other hand, not possible for a user to specify exactly the location in the system's data structure of the instance desired of the term. The user must be allowed to put his query in terms that are meaningful to him. This is possible because he can specify which e-message it is that contains the term-instance he wants. For example, he may form the *e-query*:

Prodgrp K, article # 325, order, ?, date 720911

by writing the e-message with a question mark replacing the term he wants. The system would reply: "customer # 127" [see Fig 4(a)]. Similarly, if he writes the e-query as "e-message 5, quantity, ?," the reply would be "7 gallons."

The user may know the content of e-message 5, but he may not know that that message has been given the name "e-message 5." He may then replace "e-message 5" by its content, in which case "e-message 5, quantity, ?" is replaced by the equivalent query:

Prodgrp M, article # 12, order, customer # 127, date 720911; quantity, ?;

and the answer would be: "7 gallons" [Fig. 4(a) or, equivalently, Fig. 4(b)].

REFERENCES

1961. Carnap, R. *Introduction to Semantics and Formalization* of Logic. Cambridge, Mass.: Harvard University Press.
1964. Bar-Hillel, Y. *Language and Information*, Selected Essays. Reading, Mass. : Addison-Wesley.
1972. Langefors, B. *Theoretical Analysis of Information Systems*. Philadelphia: Auerbach. (Also Lund, Sweden: Studentlitteratur.)
1975. Langefors, B., and B. Sundgren. *Information Systems Architecture*. New York: Petrocelli/Charter.

B. LANGEFORS

INFORMATION PROCESSING

For articles on related subjects see **ACCESS METHODS**; **ARTIFICIAL INTELLIGENCE**; **INFORMATION AND DATA**; **INFORMATION RETRIEVAL**; and **SYMBOL MANIPULATION**.

For articles on related terms see **ADDRESSING**; **AMERICAN FEDERATION OF INFORMATION PROCESSING SOCIETIES**; and **INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING**.

Information processing might, not inaccurately, be defined as "what computers do." In fact, the broadest professional organizations concerned with computer science are named the American Federation of Information Processing Societies, and the International Federation for Information Processing, respectively.

For information to be processed by a computer or by any other information processing system, it must somehow be represented or symbolized. Hence, information processing is essentially synonymous with symbol manipulation, and the entire discussion in this Encyclopedia of symbol manipulation could be readily retitled "information processing." In this article, we will approach the topic of information processing in a somewhat more philosophical, less technical, vein than in the article Symbol Manipulation.

The phrase "information processing" is often used in preference to "computation" or "data processing," to emphasize the generality of computers—the fact that they are in no way limited to manipulating just symbols that designate numbers, but can operate in any domain, numerical or non-numerical, where information is represented in symbolic form. The term "information," in turn, carries allusions to the Shannon-Wiener theory of selective information, which emphasizes the role of symbol structures as designating one particular state of affairs out of some larger set of possible states. Thus, if we are dealing with the class of flowers, the symbol "rose" conveys the information that we are concerned with a particular subclass of that class.

Information has other aspects besides the selective aspect emphasized in the Shannon-Wiener theory. However, this selective aspect is closely connected with the way in which information is used by information processing systems such as computers. Information processing systems are capable of executing a *conditional branch* or transfer operation. The conditional branch operation detects which of sev-

eral different states of affairs prevails (e.g., which of several symbol structures is stored in the working memory of the computer), and sends the subsequent computation along different paths depending on which state of affairs is detected. Thus, as the basis of the selective information available to it, the information processing system behaves in a selective, or informed, fashion.

The use of selective information by conditional branch processes lies at the root of everything complex or clever that a computer can do. In the simplest case, the conditional branch detects when an iteration is done (when the adding of a column of figures has been completed), and transfers control to the next process. (It was with this use in mind that Babbage first invented the conditional branch.) In more complex situations, conditional branching processes enable information processing systems to engage in all kinds of intelligent problem-solving behaviors (whether the intelligence be artificial or natural).

Effective information processing of ten depends crucially on substituting a high degree of selectivity (that is, a high degree of dependence on selective information) for a large amount of brute-force search through immense spaces of possible alternatives. Popular accounts of the computer often emphasize the impressive speed of its basic arithmetic processes and the vast number of computations it can perform in a short time. In actual fact, apart from "number crunching" applications, the arithmetic speed of the computer is far less important than its capability for selectivity, using information interpreted by the conditional branch processes.

Empirical research on human chess-playing skill, for example, shows that masters do not explore more alternatives than ordinary players and probably do not even usually look more moves ahead. Instead, their superior performance almost certainly rests on looking at the *right* things—i.e., using information effectively to explore selectively. Similarly, artificial intelligence applications of the computer, whether for chess playing or in other tasks, always require the use of information to behave selectively, rather than rely primarily on the speed of the machine to carry out extensive searches.

We can illustrate this trade-off between selectivity and speed in information processing by two examples: programs for retrieving information from large stores, and programs for solving problems.

information Retrieval. Whenever we have a large store of data—say, a set of customer records

INFORMATION PROCESSING

-it becomes expensive to search the entire store sequentially to find a particular piece of data. We would like, instead, to be able to go directly to the point where the data is to be found and to extract it without a lengthy search. A memory that allows us to do this is often called "random access." A better description for it is "addressable, direct access," for there is nothing random about the way in which we approach it. The store is to be **addressable** so that each record in it can be designated, or pointed to, by a symbolized address (name). It is to have **direct** access so that the information processor can be switched to read the desired record directly, once its name is known, without requiring a search.

Now it is well known that to select a particular item from a set of n items requires approximately $\log_2 n$ binary switching operations. Suppose we have a store of 64 records. Since $64 = 2^6$, we can use strings of 6 binary digits each (e.g., 100110) to provide distinct addresses for the 64 records. An appropriate switching device would have to perform six switching operations—one for each digit—to select a desired record. With such a system, the number of switching operations required to select a record increases only with the logarithm of the number of records—6 binary operations, as we have seen, for 64 records; 10 operations for 1,024 records; and 20 operations for more than a million records.

An unindexed book (or a nonalphabetized encyclopedia) frustrates human information processors because it provides no means to find a desired item of information without linear search. Thick books are proportionately more frustrating in this respect than thin books. A good index converts the book into an addressable, direct access store. The cost of retrieving an item can now be expected to increase only with the logarithm of the size of the book.

Problem Solving. To illustrate how information permits selectivity in solving problems, we will examine a trivially simple example.

How do we use an information processor to solve this arithmetic equation:

$$5x + 3 = 2x + 7$$

If we depended only on the processor's speed, we might try a simple **generate-and-test** method: Generate various values of X and substitute them in the equation; then test whether the two sides are equal. The futility of this approach is evident as soon as we ask, "Over what class of values shall we generate-integers, rational numbers, real numbers—and in what order?" Of course a very fast com-

puter might solve such problems in a reasonable time, if only problems involving small numbers were presented and possible solutions involving fractions with small numerators and denominators were generated first.

A second approach might be to write the equation as

$$5x + 3 - 2x - 7 = 0.$$

Then we could generate a possible solution and test to find if it gave a positive or negative value to the left side. If the values were positive, this information, communicated to the generator, could cause it to next generate a smaller possible solution; if the values were negative, a larger solution. In this way, the feedback of information could guide the generator to the correct solution by a process of successive approximations. Computational algorithms that employ successive approximations use information in this general way to reduce the amount of search.

Of course a far more effective way to solve the original equation is to observe that the solution is an expression of the form $X = K$, with no constant on the left side, no term in X on the right side, and X having unity as its coefficient. By subtracting 3 from both sides of the original equation, then subtracting $2X$ from both sides, and then dividing the resulting equation through by 3, we obtain the final result, $X = 4/3$, without any search whatsoever. This was accomplished by comparing the given equation with the form of the desired solution, and taking specific actions to bring it into the desired form based on the specific differences noted. Thus, when the constant 3 is found on the left side, where no constant is wanted, it is removed by subtracting 3 from both sides.

At each step, specific information extracted from the problem expression is used to choose a specific action that will alter the expression in the desired way. Since all the required selectivity is provided by the information embedded in the given symbolic expression, no search is required to find the answer. The safe can be opened, so to speak, by reading off the correct combination, rather than by spinning the dials to try different settings. Simple as it is, this example is a prototype for the most sophisticated artificial intelligence system, and contains in rudimentary form the information processes needed for carrying out **means-ends analysis**. (Means-ends analysis involves deleting one or more differences between an actual and a desired situation and then applying operators to reduce one or more

of the remaining differences as described in the algebra example above.)

A basic reason, then, why we refer to computers as information processors is that they have to not only provide us with information-by performing a numerical computation, retrieving data from a store, or in some other way-but also to respond to new information, enabling them to substitute a high degree of selectivity for speed in search as a means of solving problems.

REFERENCES

1968. Minsky, Marvin (Ed.). *Semantic Information Processing*. Cambridge, Mass. : M.I.T. Press.
Contains examples of how systems use information to guide search in sophisticated ways.
1972. Newell, Allen, and Herbert A. Simon. *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall, chap. 4.
This work discusses selective search, and describes a number of general search methods, including means-ends analysis, and their properties.
1972. Simon, Herbert A., and Laurent Siklóssy (Eds.). *Representation and Meaning*. Englewood Cliffs, N.J.: Prentice-Hall.
Further examples of sophisticated search in information processing systems that use information to guide search in sophisticated ways.

H. A. SIMON

INFORMATION RETRIEVAL

For articles on related subjects see **CURRENT AWARENESS SYSTEM**; **DATA SECURITY**; **DATA STRUCTURES**; **INFORMATION AND DATA**; and **INFORMATION SCIENCE**.

For articles on related terms see **COMPUTER NETWORKS**; **FILES**; and **INFORMATION SYSTEMS**.

Information retrieval (IR) is concerned with the structure, analysis, organization, storage, searching, and dissemination of information. An information retrieval system operates on the one hand with a stored collection of information, and on the other with a user population desiring to obtain access to the stored items. An IR system is thus designed to extract from the files those items that most nearly

correspond to existing user needs as reflected in requests submitted by the user population. A library storing books and serving a population of customers is then, among other things, an example of an information retrieval system.

For some years, the information retrieval area has been of concern to the increasing number of people interested in science and technology, in part because of the continued outpouring of potentially useful information-the production of printed materials, for example, is thought to increase yearly at a rate of about 10%—in part because of the ever-mounting costs of information generation, and in part because of the increasing technical difficulties in selectively distributing a large volume of information to a heterogeneous user population.

Conceptually, it is possible to reduce the operations of a typical information retrieval system to the following two main types: *information analysis*, normally consisting of the assignment to each stored item and to each search request of indicators designed to reflect the information content of the given item; and *information organization* and *file search* concerned with the manner in which the stored information is organized in the file and with the corresponding search procedures. Normally, a useful search strategy depends primarily on the organization of the information in storage, on the particular kind of information need expressed by the user, and on the equipment available to carry out the retrieval work.

In recent years, many of the operational retrieval services have implemented on-line operations, using console terminal devices to introduce search queries and to obtain retrieval output. In that case, the information searches may take place *interactively* in such a way that information supplied by the users during the search operation is used to obtain improved search output. Furthermore, **networks** of information centers may be created by supplying suitable connections between individual centers, thereby affording the user population a chance to access the resources of the whole network.

The establishment of information nets, capable of storing large masses of data and of making them available to vast user populations in remote locations, raises complicated legal and social problems, connected in part with the propriety of unlimited duplication and transmission of information that may be subject to legal restrictions (as is the case for patented and copyrighted information), and in part with the preservation of information privacy, where this may be warranted.

INFORMATION RETRIEVAL

Retrieval operations and techniques used in conjunction with library or text processing systems are also of interest in a variety of different information processing systems, including data management systems, selective information dissemination systems, and fact retrieval or question-answering systems.

Indexing and Content Analysis. In most operational retrieval situations, information analysis is carried out manually by using subject experts or trained indexers to assign content identifiers to information items and search requests. Such information identifiers are known variously as keywords, index terms, subject indicators, or concepts, and the search operation often consists in matching sets of keywords assigned to stored information items with keywords representing the search requests. The matching is followed by the retrieval of those items whose content indicators exhibit a sufficiently high degree of similarity to the query indicators.

A typical set of words, or word portions, indicative of the notion of "toxicity" is contained in Fig. 1. Such terms might then be assigned for purposes of content identification to documents and queries in the area of toxicity.

toxic . . . , poison . . . , lethal dose, LD, side effect, drug **allerg** . . . , drug reaction, drug sensiti . . . , **intoxicat** . . . , venom . . . , side action, side reaction, adverse effect, adverse reaction, ill effect, idiosyncra . . . , overdos . . . , overtreat . . . , intoleran . . . , **contraindicat** . . . , salicylism, goitrogen . . . , nephrotoxic . . . , neurotoxic . . . , **hyper**-vitaminosis, untoward, undesirable, deleterious, irritat . . . , irritan . . . , harm . . . , risk . . . , danger . . . , hazard . . .

Fig. 1. Terms denoting notion of toxicity that may be assigned during document and query analysis.

While the indexing practice is still largely manual, automatic indexing methods are becoming increasingly popular, particularly in a document retrieval environment where references to stored documents are retrieved in response to incoming search requests. The following types of operations are then often used:

1. Expressions are chosen from document or query texts, consisting variously of words, word stems, noun phrases, prepositional phrases, or other content units, which exhibit certain specified properties.

2. Weights may be assigned to each expression on the basis of the frequency of occurrence of the given expression, or the position of the expression in the document, or the type of entity.

3. The expressions originally assigned may be replaced by new ones, or new "associated" expressions may be added to those originally available, based on information contained in stored dictionaries, or on statistical co-occurring characteristics among the terms in a document collection, or on syntactical relations among words.

4. Additional relational indicators between terms may be supplied to express syntactical, or functional, or logical relationships among the entities available for content identification.

The end result of a content analysis procedure is shown in Fig. 2 for two typical queries in the world affairs area. In each case, the resulting query "vectors" are shown, including both term identifications and weights assigned to the terms.

The result of such an automatic indexing process is then similar to that previously outlined in that each stored item is identified by a set of terms representing information content. In operational systems, the automatic indexing practice is still largely restricted to the analysis of document *titles* only—the resulting search products being called "permuted" title indexes or "keyword in context" (KWIC) indexes. However, as larger text portions are made available in machine-readable form, the content analysis will extend to abstracts, summaries, or full texts, with results equivalent to, or exceeding in effectiveness, those now obtainable in manual systems. An example of a KWIC index is given in the article on that subject.

File Organization and Search Strategies.

Several classes of file organizations are commonly used, the simplest of which is the *serial file*. Here, no subsets of the file are defined, no directories are provided affording access to any subsections of the file, and no particular file order is specified. A search is then performed by a sequential comparison of the query with the identifiers of all stored items. Such a serial file organization is most economical in storage space, since no overhead is incurred for the storage of directories or links between items; furthermore, access is equally con-

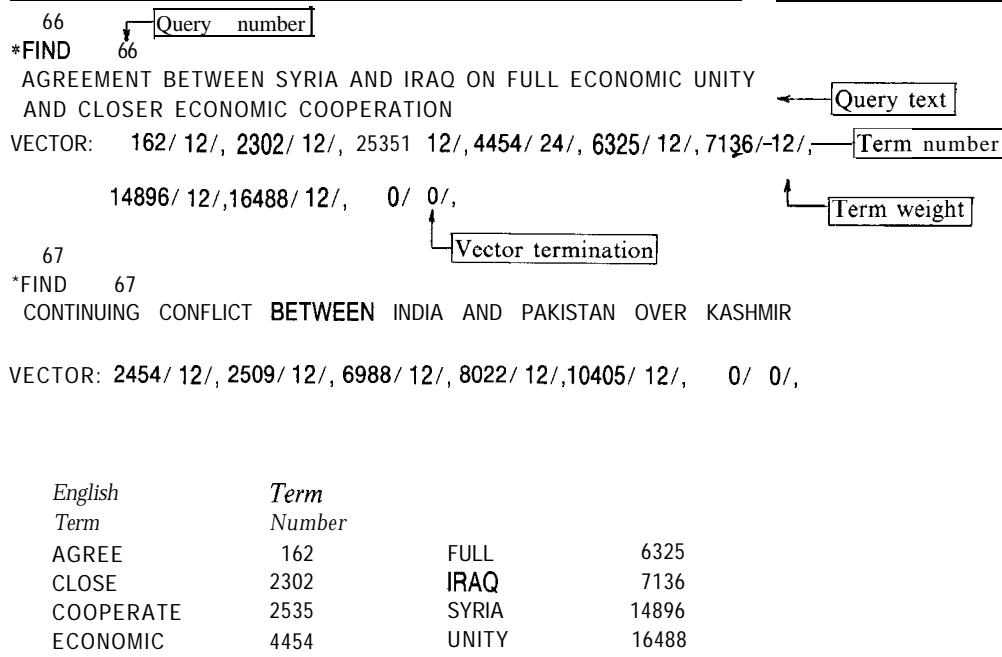


Fig. 2. Typical analysis query vectors.

venient with respect to all keyword classes such as document authors, dates of publication, or content indicators. Unfortunately, a sequential search operation is time consuming and is thus unusable if search output is expected rapidly.

An equally small storage overhead may be incurred in the *computed-access files*, where the stored information is grouped into sets of items mathematically related in some way. In this case, a computation is performed on the set of terms used for accessing, and the "hashed" result of the computation is transformed into one or more storage addresses corresponding to the locations where the requested information may be stored. The search time is very small for computed access files, and no directories may be needed in addition to the main file. However, it is difficult in practice to construct good hashing functions that produce few collisions between distinct items mapping into the same storage address.

Chained files are characterized by the fact that all items exhibiting a given common identifier are "chained" together by appropriate links, or pointers; a directory normally provides access to the first item in each chain, and the file is searched by following the pointers within the individual chains. Chained Files provide faster access than do serial files, but considerable storage overhead may be incurred to

store pointers and directories, and a problem arises when the chain lengths become excessive for certain terms.

The best known and most universally used file organization in information retrieval is the so-called *inverted file*, where a large inverted directory is used to store for each applicable keyword or content identifier the corresponding set of document or item identifications and locations. The file is thus partitioned into sets of items with common keywords, and a search in the document file is replaced by the directory search. Since only small portions of the directory need to be accessed for any given query, acceptable search times are generally obtainable. For this reason, inverted files are currently used with almost all operational on-line retrieval systems.

Inverted file organizations are advantageous in a static environment where the set of terms usable for content identification is not subject to many changes, and where access to the complete term set pertaining to a given stored item is not normally required. In a dynamic situation where changes are made to the content indicators attached to queries and documents, a *clustered file* organization may be preferable. In a clustered file, items that exhibit similar sets of content identifiers are automatically grouped into common classes, or clusters, and a search is performed by looking only at those clusters

that exhibit close similarity with the corresponding query identifiers. A clustered file produces fast search output, and the file-updating operations are relatively easy to implement.

Retrieval Operations. An automatic retrieval system generally makes a collection of machine-readable records available to a population of users that may be remotely located. Normally, the system is not constructed for a single identifiable purpose, but can accommodate many types of user queries. Searches may be conducted *off line*, in which case a sequential file search may be utilized to obtain responses within several days, or weeks, from the time of query submission; alternatively, an *on-line* search can be carried out directly from a terminal device, using an inverted file organization.

Initially, a user may submit a query statement, which is then transformed-manually or automatically-into a set of terms acceptable to the system; appropriate indicators or connectors may also be defined to express relationships among sets of terms. For example, a request covering "tissue culture studies of human breast cancer," may then be transformed into a statement of the form:

$$\left\{ \begin{array}{c} \text{Breast neoplasm} \\ \text{or} \\ \text{Carcinoma, ductal} \end{array} \right\} \text{ and } \left\{ \begin{array}{c} \text{H u m a n} \\ \text{or not} \\ \text{(any term} \\ \text{indicating} \\ \text{animal or} \\ \text{disease)} \end{array} \right\}$$

$$\text{and } \left\{ \begin{array}{c} \text{Tissue culture} \\ \text{or} \\ \text{Culture media} \\ \text{or} \\ \text{Chick embryo} \end{array} \right\} \text{ and English}$$

If an on-line console search is used, various optional displays may be available to help the user in obtaining acceptable search output. Thus, tutorial sequences may be included to inform the operator about the features of the system; displays of the available term vocabulary may be used during the generation of the query statement; finally, displays of previously retrieved information-i.e., titles or abstracts of items retrieved earlier-may help the user in constructing improved query formulations.

Such **feedback operations** are particularly helpful in obtaining more effective retrieval output.

A diagram describing a feedback retrieval system is shown as Fig. 3, and a typical on-line search protocol is given in Fig. 4.

Retrieval failures may be due to the analysis and indexing policy-i.e., the assignment of too many, or too few, or of a number of incorrect content indicators-or to the indexing language itself (i.e., to the type of vocabulary available for assignment to queries and stored information items); or to the search strategy used; or finally to problems arising during user-system interaction. The use of natural language indexing systems may ease some of the restrictions inherent in a controlled indexing language in that it creates many diverse avenues for obtaining access to the stored information; on the other hand, new problems may be introduced by ambiguous or nonstandard uses of the vocabulary. Many of the retrieval problems arising in standard systems from the lack of appropriate user-system interaction are eliminated in modern real-time search systems.

In addition, the **networks of information systems**, which are starting to be created, may relieve the inadequacy of local data banks, provide access to a greater variety of services, and furnish economy and improved use of technical competence. A simplified information network is shown in Fig. 5.

The question of **information** privacy, involving the right of individuals to obtain access to a given piece of information under specified conditions, is most complex, and no solution acceptable to all user classes is likely to emerge soon. On the other hand, it is relatively easy, at least conceptually, to provide *file* security by implementing any given set of privacy decisions. Such security measures may be data independent in the sense that a decision to provide access does not depend on the stored data, but only on the identity of the user and the type of file being manipulated; alternatively, the authority to access may be data dependent. Elaborate systems of user authentication by means of special passwords and of monitoring devices designed to detect unauthorized access are now in use in some installations.

Retrieval Applications. The most common type of retrieval situation is exemplified by a **reference retrieval** system performing "on demand" searches submitted by a given user population. Normally, only the bibliographic information is stored for each item, including authors' names, titles, journal or place of publication, dates, and applicable keywords and content identifiers. Often, only the

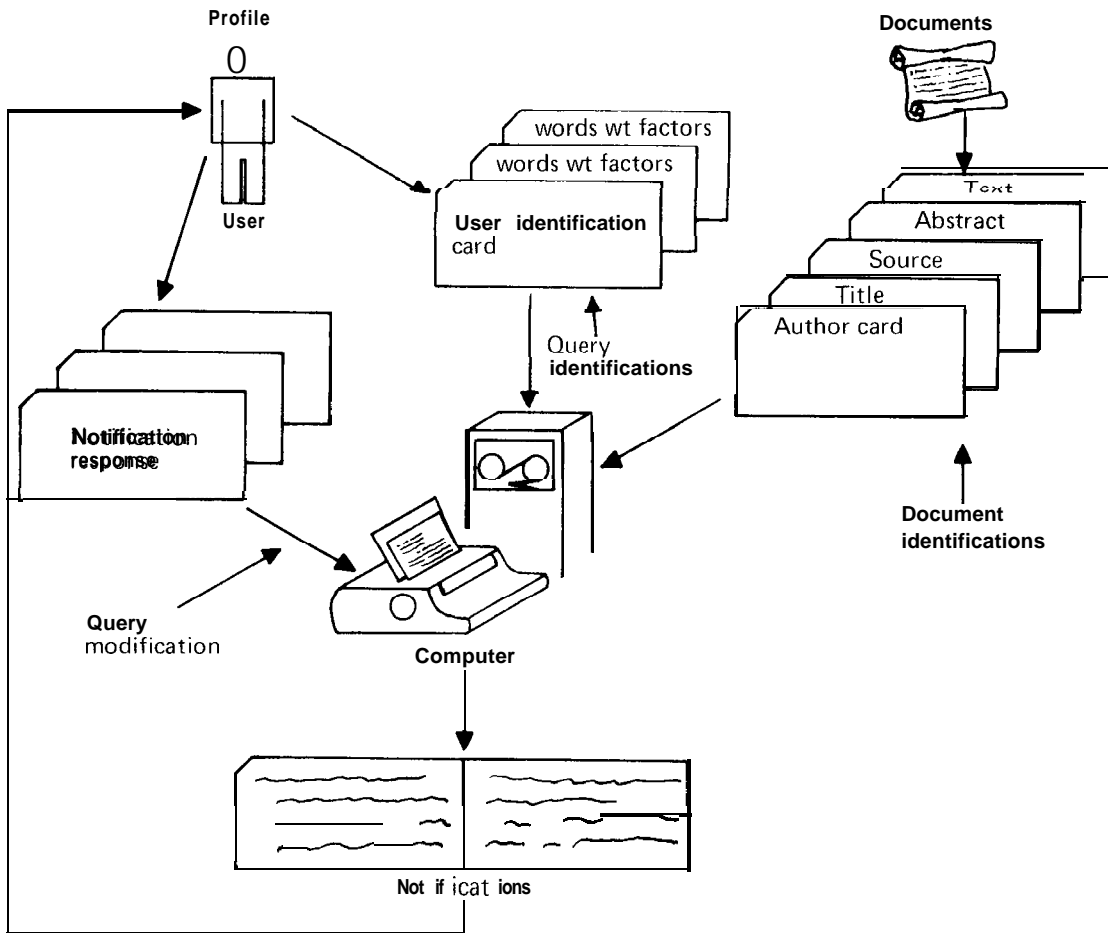


Fig. 3. Elements of retrieval system with provision for user feedback.

keywords are usable for search purposes; sometimes the words of the document titles can also be searched; less commonly, more extended text portions such as abstracts, summaries, or even full texts may be stored, in which case a text search (as opposed to a simple keyword search) becomes possible.

In any case, the responses provided by the system consist of references to the bibliographic items that match the user queries. In most conventional situations, the retrieved information is submitted to the users in no particular order of importance; an ordering in decreasing query-document similarity can, however, be obtained in the more advanced systems, which can then be used advantageously for search negotiation and feedback purposes.

A sample search output in decreasing query-document similarity order is shown in Fig. 6.

In a standard reference retrieval system, a search is conducted only when a user actually submits a search request. However, systems also exist which permanently store (and update) user "interest profiles," i.e., dummy queries that express the principal areas of interest for a given user population. Any new information items coming into the system are then periodically matched against the stored interest profiles, and the relevant output is supplied directly to each individual on a dependable, continuous schedule.

Some of the operational systems for such a *selective dissemination of information (SDI)* use response cards, submitted by the user population following receipts of a retrieved document, to update automatically the stored user profiles. Thus, as users become more or less interested in some areas, the positive or negative responses of the recipients are used to add or upgrade (or, correspondingly, to

INFORMATION RETRIEVAL

REQUEST
HYPER←KINESIS LER←ARNING D
INVALID REQUEST COMPONENTS
WILL ATTEMPT TO CONTINUE
SSYU ARE NOW IN COMMUNICATION WITH (DATA) CENTRAL.
PLEASE ENTER 10 CHARACTER IDENTIFICATION.
OSEARSAAW.
ENTER FILE, MESSAGE OPTION
EARS,S
REQUEST
HYPERKINESIS OR LEARNING DISORDERS
YOUR REQUEST IS BEING PROCESSED,
16 ENTRIES.

DO YOU WANT TO PROCESS ANSWERS; NO, PRINT OR MODIFY?
M
ADD NUMBER 002 MODIFICATION
AND SPIKE OR EPILEPTOGENIC
YOUR REQUEST IS BEING PROCESSED.
18 ENTRIES. (151)
DO YOU WANT TO PROCESS ANSWERS; NO, PRINT OR MODIFY?
N
ADD NUMBER 003 MODIFICATION
P←AND PH OR HYDROGON← ←EN ION OR BLOOD GAS
YOUR REQUEST IS BEING PROCESSED.
0 ENTRIES
DO YOU WANT TO PROCESS ANSWERS; NO, PRINT OR MODIFY?
M
INVALID RESPONSE
0 ENTRIES.
DO YOU WANT TO PROCESS ANSWERS; NO, PRINT OR MODIFY?
N
INVALID RESPONSE
0 ENTRIES.
DO YOU WANT TO PROCESS ANSWERS; NO, PRINT OR MODIFY?
PRINT2
ENTER DESIRED OUTPUT, DEVICE.
9,C
DO YOU WANT THE ENTRIES SEQUENCED? YES OR NO.
N
SET PAPER (IF NECESSARY), PRESS SPACE BAR TWICE AND TRANSMIT.

Fig. 4. On-line search protocol.

delete or downgrade) the respective terms from the profiles.

The rapid development of SDI systems is due in large part to the production and availability of a variety of tape data bases containing titles, references, and sometimes index terms of the published information in various fields.

Data management, *or management information systems*, normally provide general file processing capabilities together with user interface methods to simplify the manipulation and analysis of the stored data. In general, such systems include only simple record-keeping provisions, together with exception reporting, and output-generating capabilities based

INFORMATION RETRIEVAL

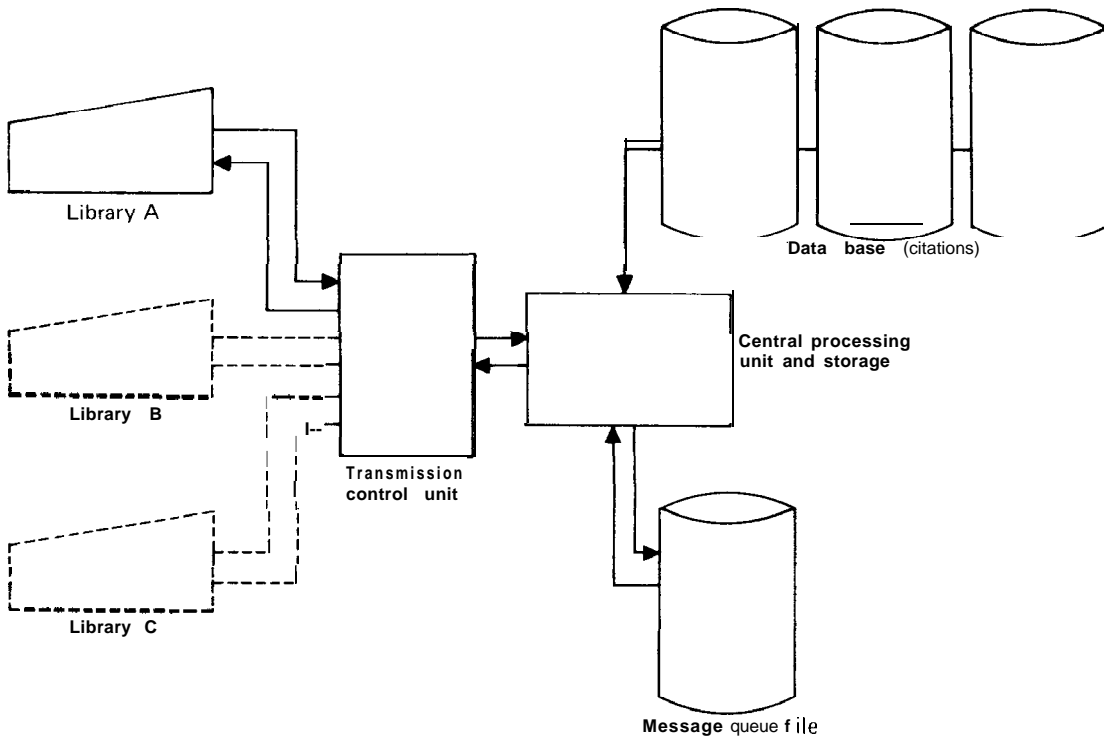


Fig. 5. Schematic diagram showing elements of a centralized network.

RESULTS OF REQUEST EVALUATION PROCESSING

EVALUATION OF REQUEST QA18TWO-D WITH 3 RELEVANT DOCUMENTS

Implication of relevance: X=yes	THE TOP FIFTEEN DOCUMENTS		RELEVANT DOCUMENT RANKS	
	Document number and identifier	Coefficient of similarity between each retrieved document and the query	Document number and identifier	Coefficient of similarity between each retrieved document and the query
Ranks of 0 retrieved documents	1 X 70X601 ENCODI	0.6155	1 70X601 ENCODI	0.6155
	2 X 02X814A NEW	0.3310	2 02X814A NEW	0.3310
	3 54X901A SELE	0.3060	8 09X206ANALYS	0.2508
	4 30X409RELATI	0.3037		
	5 60X1	0.2769		
	6 39X51 7RESEAR	0.2591		
	7 41212TOWANALYS	0.2533		
	8 X 09X206ANALYS	0.2508		
	9 67X716A NATI	0.2341		
	10 74X305THE US	0.2315		
	11 18X1122STATE	0.2226		
	12 69X813AUTOMA	0.2178		
	13 59X1208THE N	0.2138		
	14 50X418ENGLIS	0.2086		
	15 81X1107THE A	0.2059		

RANK RECAL = 0.5455 LOG PRECISION = 0.6462

NORMALIZED RECALL = 0.9789030 NORMALIZED PRECISION = 0.9139

RNK REC + LOG PRE = 1.1917 WEIGHTED NORMED RECALL + NORMED PREC = 1.8084

Fig. 6. Search output in query-document similarity order.

INFORMATION SCIENCE

on the use of statistical packages and plotting facilities.

Some management information systems also include query capabilities, permitting the user to obtain answers to certain types of submitted queries. In that case, a search-and-retrieval component of the type previously described must be included.

A final class of language processing applications of interest in retrieval are the language-understanding, or *question-answering*, systems, wherein a direct answer is expected in response to a submitted query (instead of only a set of references that may in turn contain the answers). The depth and complexity of the document-and-query analysis must be much greater in question answering than in standard reference retrieval, since a precise and detailed understanding of the queries is needed before the answers can be supplied.

Normally, question-answering systems include syntactic components based on a stored grammar and dictionary; a semantic interpreter that transforms the syntactically analyzed input into a formal query statement acceptable to the program; and finally, a deductive component that can generate responses by comparing the formalized query statement with information included in the data base.

Several experimental text-based question-answering systems have been designed, but for the moment their coverage is limited to a small discourse area and a restricted subset of the natural language. Until more is known about language understanding and semantics, the question-answering application is likely to remain a laboratory pursuit rather than a practical possibility.

REFERENCES

- 1963. Becker, J., and R. M. Hayes. *Information Storage and Retrieval-Tools, Elements, Theories*. New York: John Wiley.
- 1968. Lancaster, F. W. *Information Retrieval Systems-characteristics, Testing, and Evaluation*. New York: John Wiley.
- 1968. Salton, G. *Automatic Information Organization and Retrieval*. New York: McGraw-Hill.

G. SALTON

INFORMATION SCIENCE

For articles on related subjects see **COM-**

PUTER SCIENCE; INFORMATION AND DATA;
and **INFORMATION PROCESSING**

The term *information science* was coined to designate an interdisciplinary field initially concerned with the exponential growth of recorded scientific information. In 1950, the 81st U.S. Congress authorized the National Science Foundation to "foster an interchange of scientific information among scientists in the United States and foreign countries." Applied information science received a major impetus with the enactment of the National Defense Education Act of 1958, by the 89th Congress, which directed the National Science Foundation to establish a science information service through which the Foundation "shall (1) provide, or arrange for the provision of, indexing, abstracting, translating, and other services leading to a more effective dissemination of scientific information, and (2) undertake programs to develop new or improved methods, including mechanized systems, for making scientific information available."

In the 1960s, the thrust of applied information science focused primarily on the handling of bibliographic records and textual information in science and engineering. Two major foci of effort received considerable attention: the study of the communication processes in the communities of science and industry; and the development of techniques and systems for more efficient organization, storage, and dissemination of recorded scientific information. The term "informatics," synonymous with these two directions of effort, was coined in France (*informatique*) and popularized after its adoption by the USSR and the Soviet bloc countries; in these countries, *informatika* is considered to be a branch of the social sciences. (Terminological agreement is by no means unanimous: for example, in West Germany, as well as other places throughout western Europe, *Informatik* designates applied computer science.)

More recently, the preoccupation of applied information science with the control of recorded information and communication in the scientific sector has been broadened to encompass concern with information handling in other professions as well: management, education, medicine and health care, government, law, the military, and others. The initial premise of applied information science—that the cost effectiveness of scientific and engineering work can be raised by improving the communication among its practitioners—has been formulated into a broader assumption: that the cost effectiveness of the human information processes which characterize

these professions (e.g., problem solving, decision making, learning, etc.) can be significantly improved through their formalization and gradual delegation to symbol processing machines.

From this assumption, present-day information science and its professions derive their current social mission and long-term objective: the design of information processing systems that augment man's mind and purposeful activities. The significance of the social mission of information science lies in its extending man's historic concern with the efficiency and effectiveness of physical processes into the domain of the symbolic processes of the human mind. So formulated and interpreted, information science subsumes or provides linkages among directions and aspects of other disciplines and professions, including those of applied computer science. Indeed, to the extent that both computer science and information science share these logical aspects of an engineering discipline (an interest in the design and use of information processing engines and systems), they are considered by many to be synonymous.

As reflected in its principal review publication (*Annual Review Of Information Science and Technology*) and the programs of its professional societies (in the U.S., the American Society for Information Science), the character of recent information science has been that of a social science and/or an engineering science (technology). Increasingly it is realized, however, that significant progress in the social mission of information science may depend on its ability to develop a natural science branch of the discipline, to be devoted to basic research on the nature and properties of "information" as a fundamental phenomenon, and on primitive information processes. Such a realization motivates a growing number of academic departments in information science, the first of which was established in 1963 at the Georgia Institute of Technology, under the sponsorship of the National Science Foundation. Recently, the term "informatology" was proposed to distinguish the basic science branch of information science from its social and technological orientations.

As a basic science, information science has only begun its search for content and structure. The main direction of this incipient effort in the United States, the USSR, and western Europe is that of semiotics, the study of sign phenomena. (Signs are entities that signify some other thing, called the "object" of the sign, and can be interpreted by a sign interpreter.) This direction includes investigations of the static structure of signs-as represented by fields such as semantics, information theory, and complexity the-

ory-and the study of dynamic sign processes (semiosis) that transfer or transport sign phenomena. In this setting, information science is of **metadisciplinary** import, due to the semiotic nature of the nonphysical sciences (linguistics, psychology, sociology, history, and others) in which the essential phenomena studied are sign phenomena.

REFERENCES

1966. Cuadra, C. C. (Ed.). *Annual Review of Information Science and Technology*. Chicago, Ill.: Encyclopedia Britannica, vol. I.
1973. Debons, A. (Ed.). *Challenges to the Development of a Science Of Information: Proceedings of the 1972 NATO Advanced Study Institute in Information Science*. New York: Dekker.

V. SLAMECKA AND C. R. PEARSON

INFORMATION SYSTEMS

For articles on related subjects see **COMPUTER SYSTEMS; DATA BASE AND DATA BASE MANAGEMENT; DATA PROCESSING; INFORMATION AND DATA; INFORMATION PROCESSING; MANAGEMENT INFORMATION SYSTEMS; and PROCESSING MODES.**

For articles on related terms see **CONTROL APPLICATIONS; FILES; INPUT-OUTPUT DEVICES; MEDLARS/MEDLINE; MEMORY: Auxiliary; and STRING.**

An information system can be defined as a collection of people, procedures, and equipment designed, built, operated, and maintained to collect, record, process, store, retrieve, and display information. An information system may utilize various technologies; Sage (1968) describes the historical development of information systems in organizations from **Babylonian times**. In this article, only systems that contain digital computers as integral parts are considered; sometimes these are called **computer-based information systems (CBIS)** to distinguish them from earlier (i.e., manual) systems.

Information systems (Fig. 1), as defined above, accept (as inputs), store (in files or a data base), and display (as outputs) strings of symbols that are grouped in various ways (digits, alphabetical characters, special symbols). Users of the information systems attribute some value or meaning to the

INFORMATION SYSTEMS

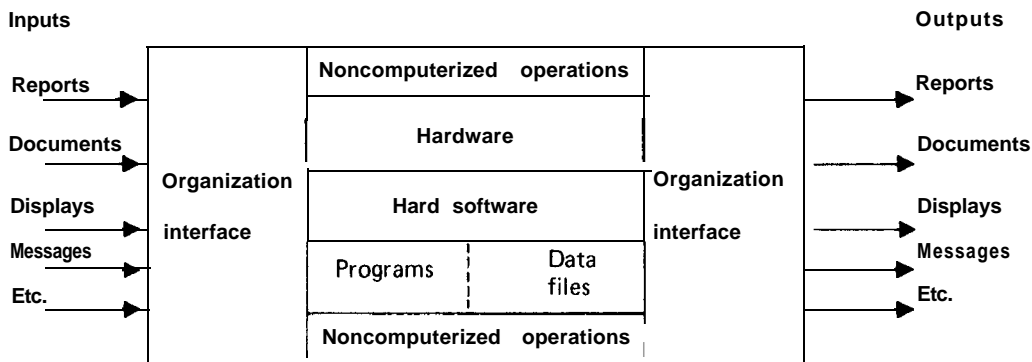


Fig. 1. Diagram of an information system.

string of symbols. Sometimes a distinction is made between the mechanistic representation of the symbols, which is called "data," and the meaning attributed to the symbols, which is called "information." A given output datum, under this definition, can result in different information to different users. In this article, the emphasis is on the common characteristics of systems rather than on the meaning attached to the output. The term "Information Processing System" (IPS) is perhaps more appropriate. It will be considered synonymous with "data processing system."

Structure. The information system itself consists of the expression, by an appropriate method, of a set of rules by which (1) the contents of the output are determined from the contents of the input and the contents of the data base, and/or (2) the contents of the data base are modified by the contents of the input. Physically, this may be viewed as shown in Fig. 1. First there are machines, or hardware, of which the most important is the CPU (Central Processing Unit) and various input and output devices such as terminals, card readers, printers, etc. Next is a set of software, including operating systems, utility programs, data base management systems, etc. In addition, there are programs specially prepared for the particular system, frequently known as the application software, which is normally prepared in some higher-level programming languages. The data is stored on auxiliary memories, such as disks, tapes, and bulk core.

Classification of Information Systems.

Information systems may be classified in various ways for various purposes. One method of classification is by the application area, but a more useful classification is by type of service rendered. The

following are among the most important classes:

1. *Computing service* systems that provide a general computing service to a number of users. Common examples are university computing centers, computing centers in research institutions, and commercial time-sharing services.

2. *Information storage and retrieval* systems designed to store data (or documents) and retrieve it in response to queries. An example is the medical information retrieval system MEDLARS.

3. *Command and control* systems built to monitor some given situations and provide a signal when predefined conditions occur. An example is the Ballistic Missile Early Warning System (BMEWS).

4. *Transaction processing* systems designed to process predefined transactions and produce predefined outputs as well as maintain the necessary data base. An example is an order-entry billing system.

5. *Message switching* systems that route messages over transmission lines from a point of origin to destination.

6. *Process control* systems designed to control physical processes by monitoring the conditions and signaling appropriate action to the machines. Common examples are systems to control chemical processes and oil refineries.

A summary of the inputs, data base contents, and outputs for these six systems is given in Table 1. Each of these types has certain characteristics that affect the structure of the system, the measures of performance which are appropriate, and the process of designing, building, and operating the system. Many systems in existence today have features from more than one type, and may be considered mixtures of the basic types.

Table 1. Typical inputs, data base contents, and outputs by types of system

Type	Input	Data Base	Outputs
Computing service	Both programs and data supplied by users	Created by individual users for their own purposes. System maintains minimal data base for control and allocating charges.	Specified by users for their own purposes.
Information storage and retrieval	Determined by system designers on basis of what is relevant to inquiries to be answered.	Contains all input received.	Produced in answer to user inquiries.
Command and control	Obtained from sensors and monitors	Built up from data received by inputs.	Warning and action notices obtained by periodic processing of inputs and data base.
Transaction processing	Predefined transactions.	Contains all data necessary to process transaction and produce outputs.	Specified by system designer to accomplish system objective.
Message switching	Messages.	Minimal. Contains data on status of nodes in network.	Messages sent to specified location.
Process control	Obtained from sensors and monitors.	Status of all processes under control of systems.	Signals to control operator of physical devices.

The users of systems may be geographically distant from the physical hardware. Users initiate different types of requests or jobs to be processed. The system has a number of different types of resources, and may have more than one of each type. Any given request or job may need more than one type of resource, possibly given in some order. There are different ways of organizing the resources to accomplish the requests, and systems may be therefore classified by the type of system organization.

BATCH OR SEQUENTIAL PROCESSING. Requests are grouped into batches on the basis of common processing requirements, and each batch is processed as a unit, usually at a predetermined time. The individual user therefore gets his results at the conclusion of all operations on the batch in which his request is included.

STORE AND FORWARD. Each resource has a queue, consisting of the jobs that require that resource. When a job is finished at that resource, it is sent to the queue at the next resource needed, and the next job in the queue is processed. The user gets his result when all the operations on his job have been performed.

IN-LINE OR RANDOM PROCESSING. Jobs are selected for processing according to some priority scheme; once a job has been started, it is processed completely through to the final result. All the necessary files in the data base are updated.

INTERACTIVE. The user communicates with the computing facility via terminals, and his requests are processed as they arrive. He gets quick responses, which he may use to prepare his next input. In order to accomplish this, it is usually necessary to provide some method of time sharing, unless the system is dedicated to a single user.

REAL TIME, OR ON LINE. When a request is received, it is acted on usually by the in-line processing method so as to provide a response within a given time period. This differs from in-line processing in that feedback is used to control subsequent inputs and in that the demand on response time is stricter.

Common Features of Information Systems. The various classifications described above are useful in identifying common features of systems that may appear in more than one type. All six

INPUT-OUTPUT CONTROL SYSTEMS

systems have certain features in common, which have important implications:

1. Information systems have to be designed, constructed, operated, and maintained. This is a nontrivial task and has led to the need for methods of system development, operation, and maintenance. An introduction to the topic is given by Benjamin (1971). A survey of current practice appears in the *EDP Analyzer*. Software engineering is a discipline emerging as a partial response to this need.

2. In the development and operation of information systems, both the programs and the data base are important in the benefit/cost performance.

3. Because of the large cost involved in developing information systems, there is an economic need for systems to share hardware, files, and programs.

4. The systems tend to be large and costly to develop, operate, and maintain. This arises because of economies of scale involved in larger hardware and in economies of scale involved in operation and maintenance of systems.

5. The systems involve man-machine communication at various levels, and problems of design and operation include both problems of communication among individuals, of communications with the machine, and of the communication among the various units of the machine. Therefore, documentation is an important aspect.

6. The uses of the systems and the technology on which the systems are developed are continuously changing, as are the organizations using them; consequently, the systems themselves are seldom if ever static.

Information systems are expensive to develop and to operate; consequently, analyses to determine whether they are serving the desired needs of users, and the measurement of their performance, are receiving considerable attention. Performance evaluation must be considered at a number of levels. At the top level, the value of the output of the system to the organization that supports it must be determined. Once these specific outputs have been justified, the performance of the physical system in achieving these outputs must be measured. This performance is a combination of the performance of programs, software, and the hardware equipment itself.

REFERENCES

1968. Sage, S. M. "Information Systems: A Brief Look into History," *Datamation* (November), pp. 63-69.

1971. Benjamin, R. I. *Control of the Information System Development Cycle*. New York: John Wiley.

1973. *EDP Analyzer*, Vol. 11, No. 5 (May).

D. TEICHROEW

INPUT-OUTPUT CONTROL SYSTEMS

For articles on related subjects see **ACCESS METHODS; DATA BASE AND DATA BASE MANAGEMENT; FILES; MEMORY; Auxiliary;** and **OPERATING SYSTEMS.**

For articles on related terms see **BLOCKS AND BLOCKING; BUFFER; LOGICAL AND PHYSICAL UNITS;** and **SUPERVISOR CALL.**

One of the earliest and most fundamental reasons for the initial development and subsequent growth of operating systems concerns the handling of input/output (I/O) operations. The transfer of responsibility for I/O operations from the programmer to the operating system has been undertaken for several reasons. First of all, the construction of code for handling I/O is one of the more difficult aspects of programming a computer. By not requiring a programmer to know the details of programming I/O operations, computing services have become accessible to a greater number of people. Secondly, as assemblers, compilers, sort packages, and other utilities became available, it was necessary that:

1. Each of these utilities be provided with I/O services.
2. User programs not be permitted to write into areas where these utilities or their work spaces were stored.

A common set of I/O routines could be used by all system facilities (and user programs, too), thus saving duplicated effort. Moreover, a simple, carefully debugged set of routines could provide some measure of protection against destruction of important files of data. The problem of accidental destruction of stored data was further compounded in operating systems that permitted users to construct and maintain private files of programs and/or data. In such systems, the denial of direct I/O capabilities to the user became even more important.

For all of these reasons, the handling of I/O operations has become almost exclusively the province of the operating system. More specifically, it has become the province of the input/output control system (IOCS) portion of a computer operating system.

Programmer Communication with the IOCS. Typically, a programmer will communicate with the IOCS by calling various modules as sub-routines. The assembly language programmer will generally have available a number of predefined macros, which will be expanded into subroutine calls to IOCS modules, using predefined calling sequences. Similarly, I/O commands in higher-level languages will generally be compiled into subroutine calls to appropriate IOCS modules. In more recent systems, these requests for I/O service have taken the form of supervisor calls.

The Functions of IOCS. The global function of an IOCS is, of course, to perform I/O operations for a programmer. This function may be refined to include the following tasks:

1. Interpretation of I/O requests.
2. Execution of I/O requests, once interpreted.
3. Location of the data to be transferred and where it is to be transferred to.
4. Initialization of transfer parameters.

These four topics will be discussed in subsequent sections.

INTERPRETATION OF I/O REQUESTS. Each of the various I/O requests that a user may make (e.g., **READ**, **WRITE**, **REWIND**, **OPEN**, **CLOSE**) must be decoded and the parameters checked. This process is accomplished by an I/O request interpreter. The interpreter will check such things as (1) the name of the operation, (2) the name of the logical unit involved, and (3) the parameters specified for the operation. Once checked, the interpreter will enter the parameters into the appropriate table (to be discussed below) and initiate execution of the I/O request.

The I/O request interpreter can, in certain cases, cause a variety of actions based on the I/O request. For example, a request to read a file that has not yet been opened might cause an error condition or simply cause the open request to be generated by the interpreter. Similarly, requests to write on a read-only device, such as a card reader, can be trapped at this level.

EXECUTION OF I/O REQUESTS. Execution of

I/O requests involves various kinds of information and routines. Among the tasks that must be handled are:

1. Maintenance of correspondences between logical and physical devices.
2. Generation of physical I/O commands based on requests.
3. Coordination of peripheral activities and maintenance of status information.

Following the distinction between logical units and physical units, it is convenient to divide the portion of the IOCS that is directly concerned with I/O transfers into two parts—logical IOCS and physical IOCS. Logical IOCS will contain routines for managing data on logical units, while physical IOCS will perform analogous functions with respect to physical units. Thus, physical IOCS will contain routines for every physical I/O device attached to the computing system (actually, these routines may be shared among devices that are all of the same type, such as all the tape drives). These routines will handle interrupts from the device and control the execution of I/O transfers without regard for the logical content, format, or organization of the data being transferred. Physical IOCS will also contain routines for handling errors and exceptional conditions received from the device.

The logical IOCS contains routines that perform functions associated with the logical unit, as declared by the programmer (or as predefined by the system). Thus, the logical IOCS will contain routines to handle blocking and deblocking, perform label verification, control error handling and recovery, sense end-of-file and other exceptional conditions, etc., depending on the characteristics associated with a given logical unit. Clearly, logical IOCS will communicate with physical IOCS when transfer of data is necessary. Table 1 illustrates the division between logical and physical IOCS for several I/O requests.

Tables for Logical IOCS and Physical IOCS. As mentioned previously, it is common to share the actual routines for performing the various functions mentioned. In order that this may be done, and also provide a capability for users to change certain characteristics, the information that is particular to a given unit is usually organized into a table. The table is then passed to the particular IOCS routine as a parameter. **Two** types of tables may be distinguished: logical device tables and physical device tables.

Physical Device Tables. Each physical I/O unit

INPUT-OUTPUT CONTROL SYSTEMS

Table 1. Division of logical and physical IOCS requests.

Request	Logical IOCS	Physical IOCS
Get the next record.	Deblock the next record. If buffer empty, get next block. If end-of-reel condition and file span multiple tapes, mount next reel.	Deliver next block from device.
Find a record in a randomly accessed file.	Request index tracks. Search index to find block of record. Request block of record. Find record and deliver to calling program.	Deliver index tracks. Deliver requested track.
Store a new record in a randomly accessed file which carries an index.	Add new record to proper block if there is space. Otherwise write new record in a separate area. Update the index to reflect the new data values.	Write updated block. Write a new record. Fetch index blocks and write index blocks.

(device) will have an associated table containing information such as the following:

1. The device type and an indication of the data paths that may be used to transfer data to or from the device.
2. Status information concerning whether the device is busy, which data path is being used if the device is indeed busy, and whether the device is reserved though perhaps not busy.
3. The I/O operation currently pending on this device.
4. If the device contains storage that can be allocated and freed (e.g., the device is a disk), an indication of which areas are available.
5. The address of the routine that can construct commands for initiation of I/O transfers for this device.
6. The address of the routine that handles interrupts from the device.

7. The address of the routine that processes errors from the device.
8. Pointers to logical device tables associated with this physical device, with an indication of the currently active logical device.
9. Pointers to other physical device tables which share a data path with this physical device.

Fig. 1 gives an annotated version of a portion of a physical device table.

Logical Device Tables. The logical device table is used to keep track of information pertaining to an I/O operation on a logical device. Since 'several logical devices may share a single physical device (e.g., a disk), there may be several I/O operations outstanding on a given physical device. The current operation on the physical device is, of course, contained in the physical device table, as shown in Fig. 1. The information concerning the various logical device I/O operations will reside in the

Device Status Table (DST) Entry					
Unused	Driver name	Inst.	Entry count	Alternate channel	Primary channel
Head 1 position	Head 2 position		Exit count	Inst.	Device busy/ not busy

Explanation:

Driver name: Name of subroutine that issues physical I/O commands.

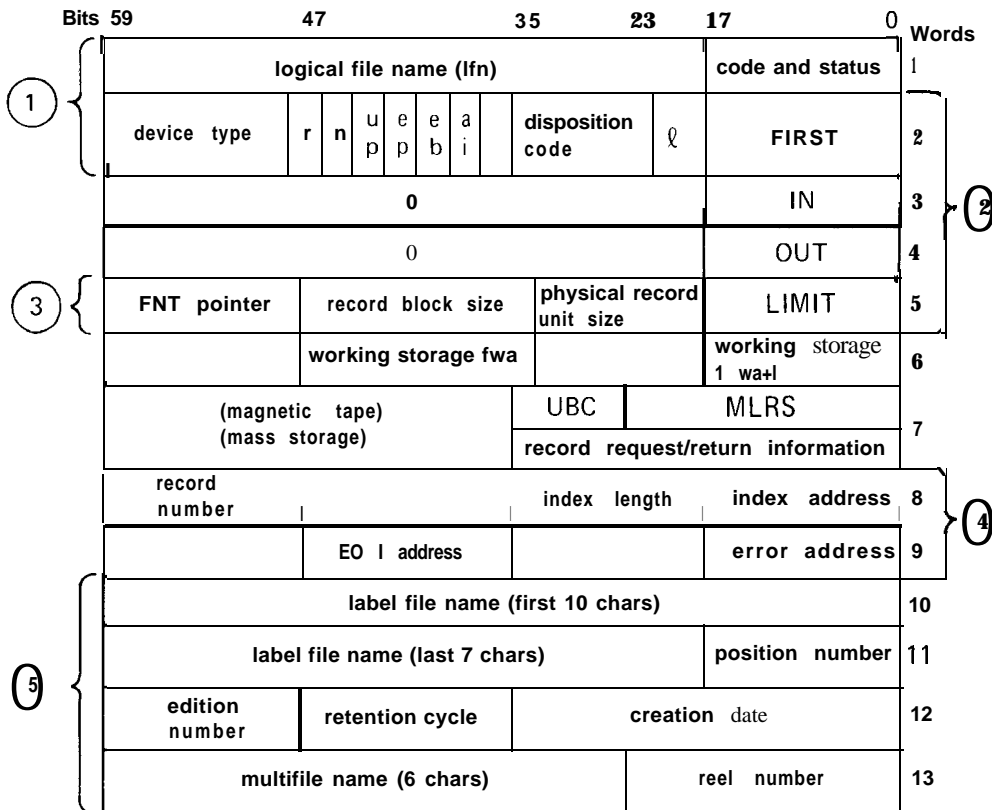
Inst: Current physical I/O instruction being executed by the driver.

Entry count: Counts the number of requests on this device,

Primary-Alternate channels: Naming of channels that can be used in conjunction with this device.

Head 1-2 positions: Status information on read/write head positioning.

Fig. 1. A portion of a physical device table. (Adapted from SCOPE 3.1 *Manual*, Control Data Corp.)



Explanation:

1. Name of the file and information concerning its corresponding physical device.
2. Buffer pointers for circular buffering.
3. Information concerning blocking factors for blocking/deblocking operations.
4. Indications of index locations for indexed sequential file organization.
5. Label information for verification and future mount requests.

Fig. 2. Annotated **logical** device table (Adapted from SCOPE 3.0 *Manual 60189400*, Rev. I, Control Data Corp.)

logical device table. A logical device table will contain information as follows:

1. The symbolic name of the logical unit.
2. The logical device type and name of the file currently attached to this logical device.
3. The logical I/O request currently pending on this logical device.
4. A pointer to the buffer(s) associated with the logical device, with indications of each buffer's status.
5. The address of the routine used for transferring data to and from buffers.

6. The address of the routine that can process interrupts, errors, and exceptional conditions for this logical device.

7. An indication of which data areas on a shared device belong to this logical device (if appropriate).

8. A pointer to the physical device table for this logical device.

9. Status information concerning the "current" address or position of the logical device, the "current" record number processed, the number of records in a buffer, etc.

Fig. 2 gives an annotated logical device table.

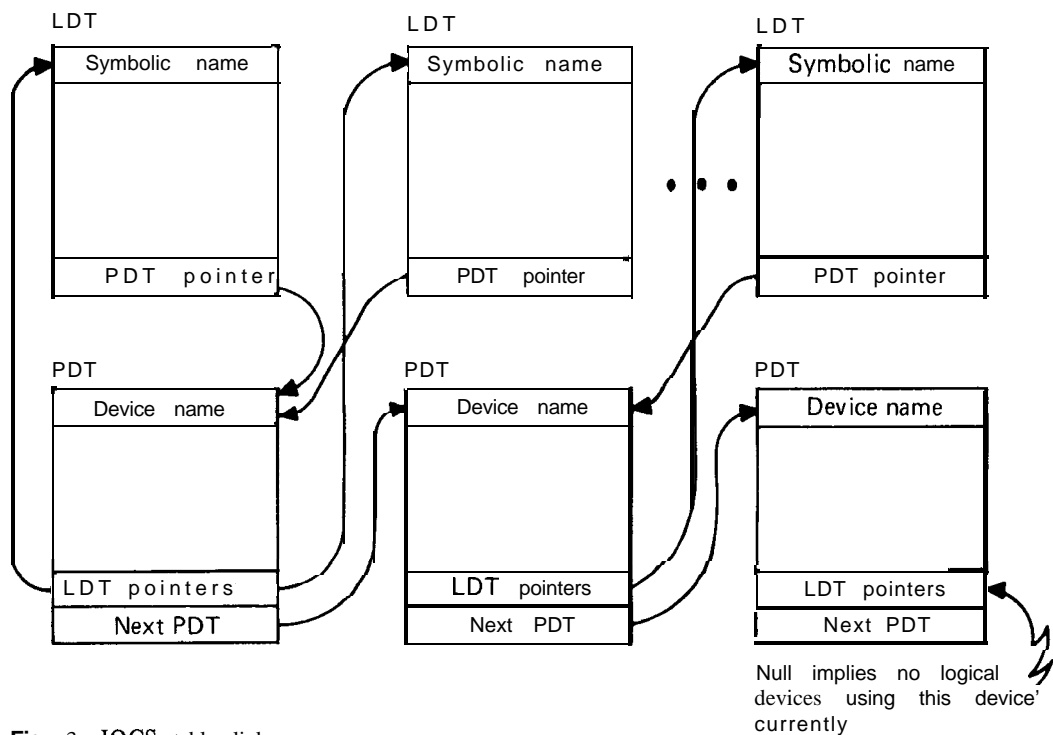


Fig. 3. IOCS table links.

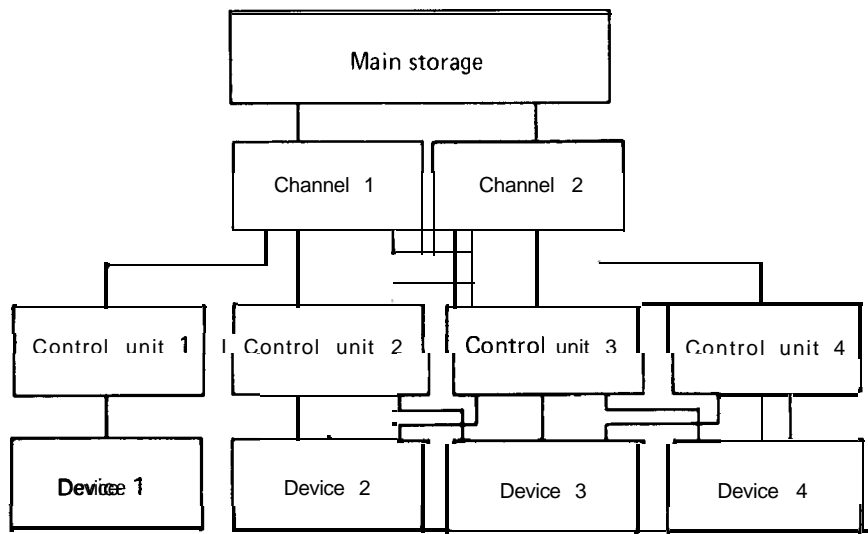


Fig. 4. Data paths to I/O devices.

It should be noted that both the logical device tables and the physical device tables contain pointers to routines that perform various functions. A programmer is typically not allowed to provide his own routines to replace those in the physical IOCS. To do so would impinge on the integrity of data stored on the physical device. However, it is common to allow programmers to supply their own routines to perform:

1. Blocking, deblocking, and buffer management.
2. Processing of exceptional conditions such as "end-of-file" or other error conditions on the logical device.
3. Label verification of nonstandard file labels (see below).

In either case, it is clear that substitution of different processing routines in place of the standard ones is simply a matter of changing pointers in the tables (and having the routines available). A programmer effects these changes by declaring that a substitution will be made and by supplying the routine. The IOCS then replaces the pointers in the logical IOCS table with pointers to these user-supplied routines.

Pointers are also used to maintain the correspondences between logical and physical devices. This may be diagrammed as shown in Fig. 3. By using the pointers from logical to physical units, it is possible to discover the physical device associated with a given logical device. Moreover, a change in logical/physical device correspondence is easily accomplished by changing a pointer in the logical device table.

Coordination of Peripheral Activities and Maintenance of Positioning Information. The scheduling and coordination of peripheral activities is an especially important IOCS function. In a large computer system, there will often exist a variety of data paths from the central processors through the data channels to the particular devices. Fig. 4 illustrates a typical situation.

Notice in Fig. 4 that a given device may be "attached" to more than one control unit and/or channel in order to form a path that can deliver data to or take data from main storage. This does not imply that data flows to or from the device over two paths simultaneously; only one path to or from a device is used at a given time. The multiple paths exist in order that devices may be kept busy as long as there exists at least one unused path to the device. The multiple paths also allow for continued oper-

ation should certain units in a data path break down temporarily. However, the IOCS must keep track of what data paths are currently in use and prevent new requests from using these paths. When a unit signals that a certain component of a path is no longer needed, the IOCS will search the pending requests to see if one can be initiated over the freed path.

In deciding on the next request to be serviced, it is convenient for physical IOCS to have information concerning the current position of read/write heads relative to the position of the data. This is particularly true with disks, which involve movable read/write heads. Requests for data near the current head position can be serviced more quickly than requests that require considerable head movement. Thus, in the scheduling of I/O operations, it is not unusual for physical IOCS to have as part of its status information an indication of current read/write head position. Using this information, it can attempt to optimize requests serviced per unit time (or some similar measure) by scheduling I/O operations based on "nearness" of data to the heads. Note also that the chain of physical device tables in Fig. 3 defines an ordering of physical devices, which can be used for deciding which of a number of devices will be started first when more than one device could be started.

LOCATION OF THE DATA AND INITIALIZATION OF TRANSFER PARAMETERS. It should be clear that before I/O requests can be interpreted and subsequently executed, the storage area that contains or will contain the data must be located and made accessible to the IOCS. Moreover, various parameters in the logical and physical device tables must be specified. The location and initialization functions are responsible for these tasks.

The location function involves routines for finding the physical devices on which the storage area to be processed resides. This storage area may or may not be directly accessible, depending on the particular computer system involved. If, for example, the programmer has attached a logical device to a tape drive on which a specified tape is to be mounted, then the IOCS must make sure that the tape is indeed mounted. This will typically involve a request to the computer system operator to mount the specified tape. It also usually involves a *label verification* routine. In order to check that the operator has indeed mounted the correct tape, a tape label in a prespecified format will usually exist on the first record of the tape. The label will contain information that identifies the tape, and the label verification routine will match the identification on the tape with the identification information given on

INPUT-OUTPUT DEVICES

the request for tape mount. Lack of a match indicates an error, and an appropriate message will be issued.

If the storage area resides on a disk or other sharable device, a somewhat different kind of location function usually takes place. There will generally exist a catalog of all files that have been created in the system, and a request to attach a logical device to one of these files will trigger a search of this catalog. The catalog will indicate on which disk pack(s) the storage area has been allocated, and a mounting of disk pack(s) onto disk drives may be necessary if the relevant storage areas are not available. Should such a mount be necessary, a verification of the mounted disk pack will take place. However, it should be noted that most interactive systems leave the available disk packs permanently mounted, so this step may not be necessary. Each disk pack will typically have a table of contents, which is essentially a collection of file labels for files on this pack. By searching this table of contents, the file is located.

INITIALIZATION. Once the data have been located, the initialization function can be executed. In order for I/O requests to be executed, various entries in the logical and physical device tables must be filled in. These parameters may be specified on a system control card or by the programmer during execution, but in certain cases they may reside with the data itself, usually as part of the file label. Thus, if it is appropriate, the initialization routines will move a copy of these parameters to the appropriate table entries.

When the file is no longer needed, a final set of IOCS routines will restore the file to a state in which it can be used at a later time. This will involve such things as marking the end of a tape, rewinding it, and informing the operator that it may be dismounted, or updating the table of contents for a file on a disk.

REFERENCES

- 1966. Clark, W. A. "The Functional Structure of OS/360: Part III-Data Management," *IBM Systems Journal*, Vol. 5, No. 1, pp. 30-51.
- 1966. Flores, I. *Computer Software-Programming Systems for Digital Computers*. Englewood Cliffs, N.J.: Prentice-Hall, pp. 22 1-322.

R. W. TAYLOR

INPUT-OUTPUT DEVICES

For articles on related subjects see **AUDIO RESPONSE TERMINAL; CARD READING AND PUNCHING TECHNIQUES; COLLATING SEQUENCE; DATA ACQUISITION COMPUTER; DATA PREPARATION DEVICES; KEYBOARD STANDARDS; MEMORY: Auxiliary; OPTICAL CHARACTER READERS; OPTICAL MARK READERS; PAPER TAPE; PRINTING TECHNIQUES; and TERMINALS.**

For articles on related terms see **CURSOR; and LIGHTPEN.**

Input is the process of translation of incoming information into electronic patterns suitable for computer processing. Output is the reverse process in which the electronic patterns are translated into a form readable by other machines or understandable by human beings. The translation process is carried out by the input and output devices of the computer system.

The most natural media for communication between a human being and a computer are those which are most natural for communication between people. For input to the computer, this would mean speaking, writing (preferably handwriting) or drawing, and movements of the hands, like pointing, etc. As for computer output, the preferred form by a human being would be hearing (spoken sentences, numbers, and/or sounds like alarm signals, etc.), reading (written messages), or seeing (drawings, graphs, or other types of pictures along with visual sensing of colors). The use of such natural I/O devices is constantly increasing as a result of the recent developments in this field, but these devices will not be discussed further in this article unless they have progressed beyond the research and development stage. However, all those now in general use, as well as some special I/O devices, are described and listed in Table 1, which also shows a few of their characteristics and some typical systems in which they may be used.

In spite of the fact that Table 1 was designed with great care, computer technology (hardware and software) as well as application fields of computer systems change rather quickly. Such changes will surely have impact upon the content of the Table 1, and it should therefore be looked upon as a general guide rather than as correct in every detail. The devices and systems in the table are cross-referenced to sections of this article in the left-hand column,



Fig. 1. CDC 3300 console arrangement. (1) Typewriter; (2) typewriter switches; (3) breakpoint switch assembly; (4) console condition switches; (5) access keyboard switches; (6) emergency off-switch; (7) step-rate control; (8) entry switches.

Table 1. Uses of Input/Output Devices in Some Typical Systems

NOTATION. For Characteristics: * = not so used. - = not available or not pertinent. (A), (N), (C), (G) = sometimes, exceptional. (C)² = R & D stage.

For Systems: Y = yes. N = no. (Y) = sometimes, exceptional. (Y)? = unknown. Y¹ = operator's console with each processor and at remote stations. Y² = R & D stage. Y³ = special systems.

class	DEVICES Groups and Subgroups	CHARACTERISTICS							Off line, con- ventional computer	On line con- ventional computer	Terminal, time- sharing system
		(1) I, O, or I/O. (2) Type: numerical only (N), alpha- numerical (A), graphical (G). (3) Use: terminal (T), conversational mode (C)									
		(1)	(2)	(3)							
1	NUMERICAL AND ALPHANUMERICAL										
	<i>Operator's control</i>										
	Console	I/O	-	A	*	*	*	N	Y	Y ¹	
	Card readers and punches										
	Readers	I	N	A	-	T	-	Y	Y	Y	
	Punches	O	N	A	-	T	-	Y	Y	Y	
	Reader /punch	I/O	N	A	-	T	-	Y	Y	Y	
	Paper tape readers and punches										
	Readers	I	N	A	-	T	-	Y	Y	Y	
	Punches	O	N	A	-	T	-	Y	Y	Y	
	Magnetic-ink and optical-character readers										
	MICR document	I	N	A	-	-	-	Y	Y	N	
	MIMR document	I	N	A	-	-	-	Y	Y	N	
	OCR page	I	N	A	-	T	-	Y	Y	Y	
	OCR document	I	N	A	-	T	-	Y	Y	Y	
	OCR journal tape	I	N	-	-	-	-	Y	Y	N	
	OMR page	I	N	(A)	-	T	-	Y	Y	Y	
	OMR document	I	N	(A)	-	T	-	Y	Y	Y	
	Printers										
	Strip	O	N	A	-	T	-	N	(Y)	(Y)	
	Digital	O	N	(A)	-	T	-	Y	(Y)	(Y)	
	Serial	O	N	A	(G)	T	-	Y	Y	Y	
	Line	O	N	A	(G)	T	-	Y	Y	Y	
	Simple keyboard	I/O	N	A	-	T	C	(Y)	Y	Y	
	Complex keyboard	I/O	N	A	-	T	C	(Y)	N	Y	
	Direct keying										
	Numerical keyboards	I	N	-	-	T	C	N	(Y)	Y	
	Touchtone phone	I	N	-	-	T	C	N	Y	Y	
	Alphanumeric keyboards	I	N	A	-	T	C	N	Y	Y	
	Special keyboards	I	(N)	(A)	-	T	C	N	Y	Y	

DEVICES USED IN SYSTEMS					
Conversation terminal, time-sharing system	Data Collection Systems				
	Remote station, single-keyboard system	Remote station, multiple-key-board with data concen-tratox	Remote station, multiple-key-board with message switching	Remote station, multiple-key-board, direct on line to computer	On line, industrial supervising or process control
Y ¹	Y ¹	Y ¹	Y ¹	Y ¹	Y
N	N	N	Y	N	N
N	N	N	Y	N	N
N	N	N	Y	N	N
N	N	N	Y	N	(Y)
N	N	(Y)	Y	N	(Y)
N	N	N	N	N	N
N	N	N	N	N	N
N	(Y)	(Y)	N	(Y)	N
N	(Y)	(Y)	N	(Y)	N
N	(Y)	(Y)	N	(Y)	N
N	(Y)	(Y)	N	(Y)	N
(Y)	N	N	N	(Y)	Y
(Y)	N	N	N	(Y)	Y
(Y)	N	N	(Y)	Y	(Y)
(Y)	N	N	(Y)	Y	(Y)
Y	Y	Y	Y	Y	Y
N	Y	Y	Y	Y	N
N	Y	Y	(Y)	Y	Y
Y	Y	Y	(Y)	Y	N
Y	Y	Y	(Y)	Y	Y
Y	Y	Y	(Y)	Y	Y
Y	Y	Y	(Y)	Y	Y

(Table continued to next page)

INPUT-OUTPUT DEVICES

class	DEVICES Groups and Subgroups	CHARACTERISTICS										
		(1) I, O, or I/O. (2) Type: numerical only (N), alpha- numerical (A), graphical (G). (3) Use: terminal (T), conversational mode (C)							Off line, con- ventional computer	On line con- ventional computer	Terminal, time- sharing system	
		(1)	(2)		(3)							
	<i>Alphanumeric visual display</i>											
	Key input checking	I	N	A	—	T	C	N		Y	Y	
	Alphascope (I/O)	I/O	N	A	(G)	T	C	N		Y	Y	
	Plasma, and others	O	N	A		T		N		Y	Y	
	<i>Audio</i>											
	Input	I	N	A	—	T	(C) ²	N		N	N	
	Output	O	N	A	—	T	C	N		Y	(Y)	
	<i>I/O in industrial (and similar) processes and systems</i>											
	Input (analog/digital)	I	N	—	(G)	T	—	N		N	N	
	Output (analog/digital)	O	N	—	(G)	T	—	N		N	N	
	<i>Special I/O devices</i>											
	Ticket vendors	O		A		T	(C)	N		Y	Y	
	Cash dispensers	I/O	N	—	—	T	—	N		N	Y	
2	GRAPHICAL											
	<i>Image input</i>											
	Facsimile I/O	I/O	N	A	G		--	N		N	N	
	Manual off-line registration	I	N	(A)	G		--	Y		N	N	
	Manual on-line registration	I	N	A	G	T	C	N		Y	Y	
	Automatic image input	I	N	A	G	T	C	N		Y	N	
	<i>Electro mechanical plotters</i>											
	Drum-type digital	O	N	—	G	T	—	Y		Y	Y	
	Flatbed-type digital	O	N		G	T	—	Y		Y	Y	
	<i>Graphical visual displays</i>											
	Graphoscopes, entry	I	—	—	G	T	C	N		Y	Y	
	Graphoscopes, I/O	I/O	N	A	G	T	C	N		Y	Y	
	TV display	I/O	N	A	G	T	C	N		Y	Y	
	<i>Microfilm I/O</i>											
	COM graph plotters	O	—	—	G		--	Y		Y	N	
	COM printers	O	N	A	—	—	—	Y		Y	N	
	COM plotter/printers	O	N	A	G		--	Y		Y	N	
	CIM	I	N	A	G		--	N		Y ³	N	

DEVICES USED IN SYSTEMS					
Conversation terminal, time-sharing system	Data Collection Systems				
	Remote station, single-keyboard system	Remote station, multiple-key-board with data concen-trator	Remote station, multiple-key-board with message switching	Remote station, multiple-key-board, direct on line to computer	On line, industrial supervising or process control
Y	(Y)	Y	(Y)	Y	(Y)
Y	(Y)	Y	(Y)	Y	Y
N	N	N	(Y)?	(Y)	Y
Y ²	N	N	N	N	N
Y	N	N	N	N	(Y)
N	N	N	N	N	Y
N	N	N	N	N	Y
(Y)	N	N	N	N	N
(Y)	N	N	N	N	N
N	N	N	N	N	N
N	N	N	N	N	N
Y	Y	Y	N	Y	N
Y	N	N	N	N	N
N	N	N	N	N	Y
N	N	N	N	N	Y
Y	(Y)	(Y)	N	(Y)	Y
Y	(Y)	(Y)	N	(Y)	Y
Y	(Y)	(Y)	N	(Y)	Y
N	N	N	N	N	N
N	N	N	N	N	N
N	N	N	N	N	N
N	N	N	N	N	N

INPUT-OUTPUT DEVICES

and additional information can be found in other articles in this encyclopedia.

Alphabetical and Numerical Input and Output Devices

OPERATOR'S CONTROL DEVICES. A console is a unit used by the operator for all manual communication with the computer. It also provides a display from the computer, generally in all or some of these forms: visual display, printed message, acoustical signals. The operator communicates with the computer by depression of switches (with specific function assigned to each of them) or by a typewriterlike keyboard. An example of a console arrangement that includes all the components listed is shown in Fig. 1.

The configuration of the console arrangement differs with the computer size and model used. For instance, in some large and fast computer systems, a line printer is used for printing information to speed up the overall performance of the system. Sometimes the console configuration can be extended upon request of the user; e.g., by adding certain features to the unit, such as a pin-feed platen to the typewriter; adding a display unit; a reference typewriter attached to smaller systems using display register and functional switches, etc. Large computer systems usually are equipped with a system console that has two CRTs and one keyboard. However, these devices may be more numerous, with several display consoles being used for controlling independent programs simultaneously.

Remote terminal stations are equipped with data station consoles to control the various I/O devices and to control communication between the data station and the central computer. A data station console generally includes a data set to connect the station to a communications channel, and also has circuits to handle automatic detection and correction of transmission errors.

CARD READERS AND PUNCHES. The punched card has been in use as a data carrier for a long time and is still used as the sole data input medium in many computer and noncomputer systems. The punched card contains data represented in the form of punched holes, which can be sensed by a variety of punched-card machines in order to carry out such functions as sorting, collating, basic arithmetic, and printing. Generally, the card has a standard size of 7.375 by 3.250 in. and a thickness of 0.007 in. It can accommodate 80 to 90 numerical digits and/or alphabetical characters. In the late 1960s, IBM introduced *System/3*, which uses a small punched

card (minicard). This has approximately one-third the area of the standard card, and accommodates up to 96 digits and/or characters and symbols.

With the advent of computers, punched cards continued to be the main data carrier for the source data input in business applications. In addition, they are often used for the transmission of user programs into the computer memory.

The function of the punched card as output from a computer-based information system has greatly diminished, apart from some applications in which the punched card has a dual function, such as a written document and as a data carrier for the machine. It is still used, however, in small punched-card computer systems and sometimes for making error corrections when using the interactive mode, and/or for amendments to programs at the program preparation stage. Also, in some batch-processing systems, short compiled programs are punched on cards.

A computer-based information system needs a wider set of symbols than that used in punched-card machines and often requires different kinds of codes. Card readers and punches, as well as the keyboards of data preparation equipment, can therefore read, recognize, punch, and print (on the surface of the card) a larger set of characters. Some readers and punches can read or punch both binary and decimal cards.

The card transport mechanism of card readers and punches is closely related to the function of the device. The basic functions are reading, punching, selecting, collating (merging, matching), interpreting (which, in the punched card machine terminology, means printing on the face of the card), **gang**-punching, reproducing, sorting, and computing. Table 2 shows the five basic types of card readers and punches, with their characteristics. Fig. 2 shows a representative card path design for the device in each category. It should be understood, however, that in each category of design, the card path may change to accommodate serial or parallel mode of operation designed for reading and punching, or checking techniques used, or the way errors detected by the checking operations are handled (e.g., selection and separation of cards with errors), or other features peculiar to the system.

Card Readers. The speed of operation of card readers is generally between 300 and 1,000 cpm (cards per minute). Also on the market are low-speed readers of 60 cpm, used in applications requiring a small punched card input only, as well as high-speed readers of up to 2,000 cpm. This latter speed seems to be the limit for safe transportation and handling

INPUT-OUTPUT DEVICES

Table 2. Basic Types of Punched-Card Input/Output Devices.

Characteristics	Basic Types				
	Reader	Punch	Read-Punch, 1 Hopper	Read-Punch, 2 Hoppers	Multifunction Unit
Functions					
Reading	Y	N	Y	Y	Y
Punching	N	Y	Y	Y	Y
Selecting	—	—	—	Y	Y
Gang-punching	N	N	Y	Y	Y
Collating, merging, matching	N	N	N		Y
Sorting	N	N	N	N	
Interpreting	N	N	N	N	Y
Computing	N	N	N	N	N
Features					
Number of hoppers	1	1	1	2	2 or more
Number of stackers	1-2	1-2	1-2	2 or more	4 or more
Speed (for fully punched 80-col. cards), cpm					
Lowest	Appr. 60	16 col./sec	Slowest function is decisive	Reader plus punch performance (if independent)	Slowest function is decisive
Average	300-500	100-200			
Top	To 2,000	500			

Note: Y, yes; N, no; —, possible.

possibilities of the mechanism. The “jams” that occur from time to time in each punched-card machine seriously affect the flow of work, and it is important to be familiar enough with the machine design to remove cards from the machine quickly, determine the number of jammed cards, and determine the damage caused.

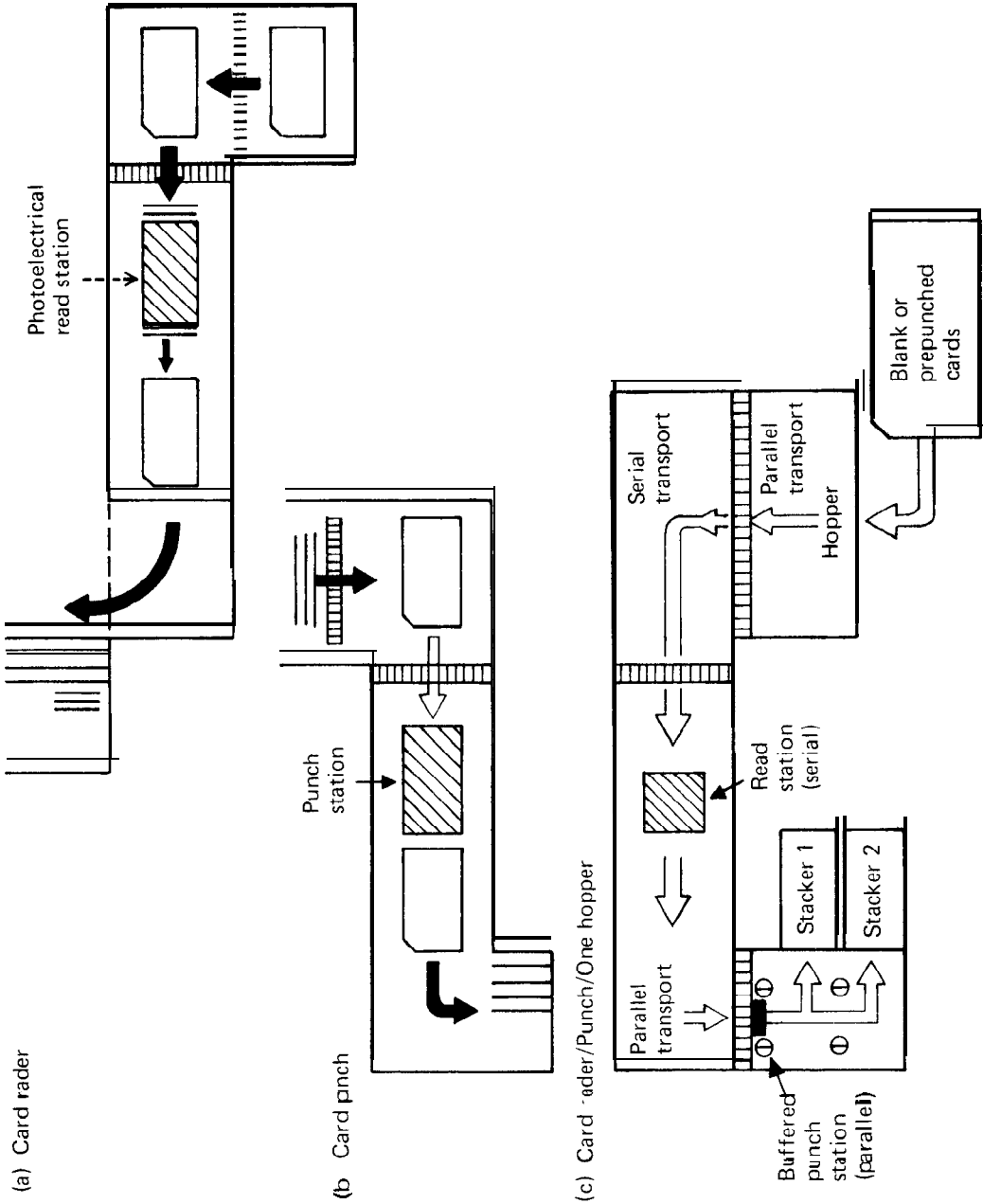
Card Punches. Card punches perform slower than card readers because of the mechanical action in punching holes in the card. Computer output devices (see Table 2) work at speeds of 100 to 500 cpm when using a parallel punching technique (e.g., row by row) or approximately 16 columns a second when slower serial punching is used.

Card Reader-Punches. The speed of operation of the card reader-punch with one hopper is determined by the chosen read or punch function. The speed of operation of the card reader-punch with two hoppers is determined by the type of operation performed. The highest performance is obtained when reading and punching take place independently, with no merging of cards from the two-card paths. Then the performance of the reader is that of

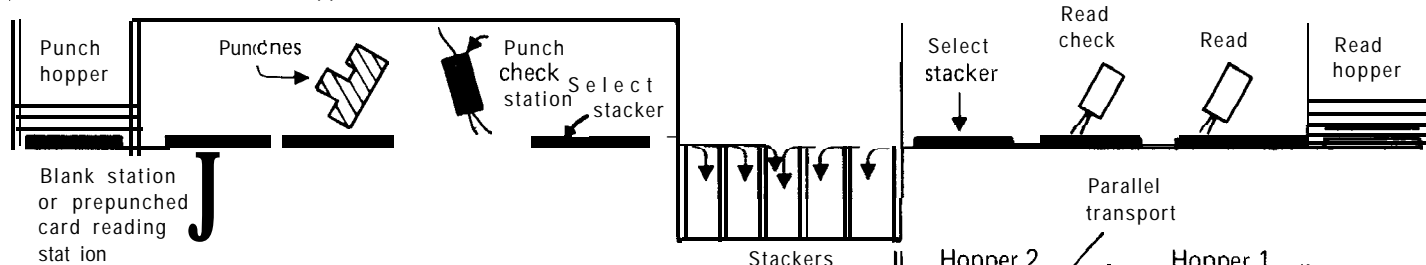
a one-hopper reader and the performance of the card punch path is the same as that of a one-hopper card punch. However, if merging of cards from both hopper paths is necessary, the speed of operation is slowed down and may sink as low as the level of the slowest (e.g., punching) path. (See Table 2.)

The last category of card I/O devices in Table 2 is represented by the multifunction card unit. Unlike devices of the four preceding categories, which are used in all computer configurations requiring reading and/or punching of cards, the multifunction card unit is used solely as an input/output device for punched-card computers, (e.g., the IBM 2560 shown in Fig. 3). The operational speed depends upon the actually performed combination of possible types of operations and upon the information content of the cards to be processed (the latter is decisive in determining the collating speed). However, the minimal speed will be equal to the speed of the slowest functional unit used in the device.

PAPER TAPE READERS AND PUNCHES. Functionally, paper tape readers and punches are similar to those of card readers and punches except that the



(d) Card reader/Punch/Two hoppers



(e) Multifunction card unit

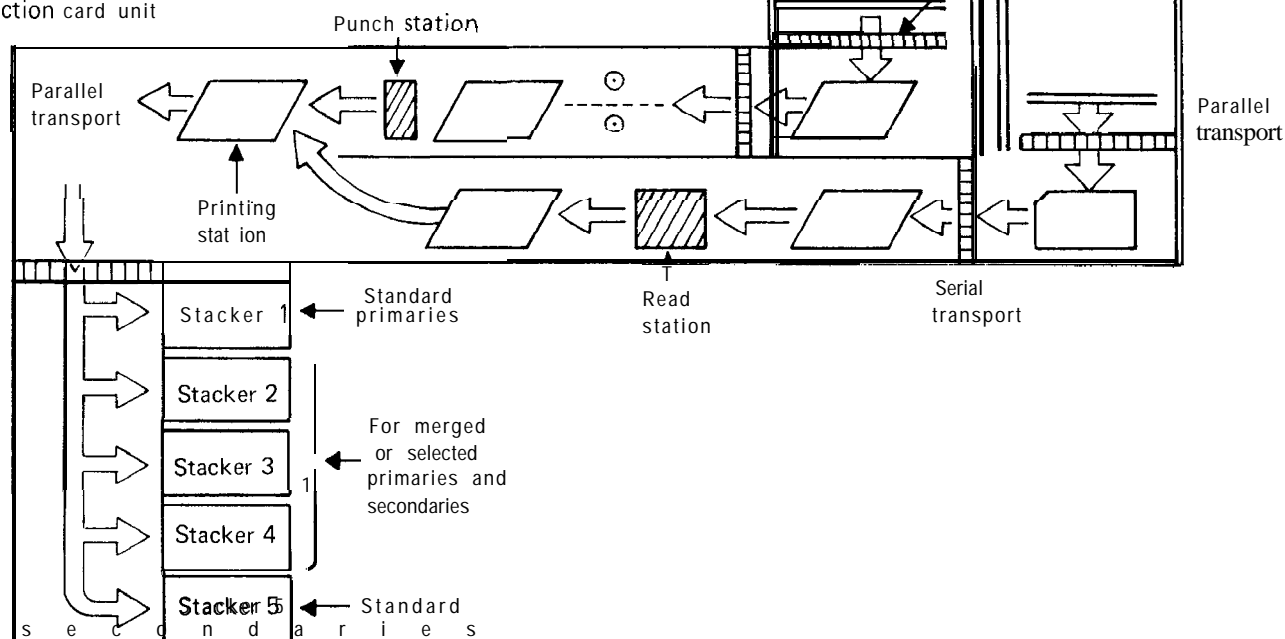


Fig. 2. Possible card-path design for each of the five basic punched-card I/O device categories. (a) Card reader; (b) card punch; (c) card reader/punch, one hopper; (d) card reader/punch, two hoppers; (e) multifunction card unit.

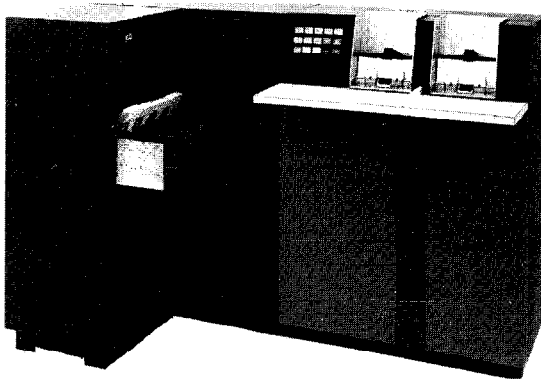


Fig. 3. IBM 2560 multifunction card unit.

information media differ. A reader translates the information punched in code on tape into the internal code of the computer and transmits the data to the computer. A punch presents coded information in the form of holes in paper tape, and can be operated manually or automatically. Automatic tape punches will be discussed here as units that are connected with the central processing unit from which they receive the information to be punched.

Paper tape readers and punches were widely used long before the advent of computers (e.g., for telegraphy), and punched paper tape has been used as an I/O medium since the earliest development of electronic digital computers. In early computer applications, the five-track paper tape used for data transmission in the telegraphic service was adopted.

Each character is recorded as a single row of holes across the width of the tape. Apart from these larger round (occasionally square) holes, the smaller

round holes (so-called sprocket holes) are pre-punched in one row along the length of the tape. These holes insure correct mechanical feeding in slow-speed readers and punches or are photoelectrically read as an indexing means for driving the tape at the correct speed in high-speed readers.

Paper Tape Readers. Paper tape readers (Fig. 4) may be classified according to speed into three categories :

1. Low-speed readers with performance from less than 1 chps (character per second) up to 50 chps.
2. Medium-speed readers with speeds ranging from 60 chps up to 500 chps.
3. High-speed readers with throughput higher than 600 chps. The top speed of at least two commercially available readers is 2,500 chps.

Some paper tape readers may be equipped with an automatic winding attachment. However, with a 2,000-chps reader, the paper advances 200 in. (approximately 5 meters) in 1 sec, assuming the individual rows are punched into the tape at 10 characters to the inch. There is therefore a possibility of damage to the tape if the winding unit gets out of adjustment. However, it is usual to wind tapes fed directly from the reader; this means that the leading end of the tape will be toward the center of the spool. The tape thus has to be rewound for further use.

In manually operated winding devices the paper tape coming from the reader is fed into a bin. Usually, the operator starts to wind the spool only after the whole tape is in the bin, putting the trailing end toward its center so that he avoids a further rewinding of the tape.

Another characteristic feature of paper tape readers is their stopping distance after a halt from full speed. This is generally one character for both lower-speed categories and at least two characters for the high-speed readers. The stopping distance depends on the manufacturer's design of the device.

Further significant characteristics of paper tape readers include:

1. **Number of tracks.** Usually the user can make a choice between a tape with round holes and from five to eight tracks and the six-track (Olivetti) square-hole tape.
2. **Checking.** Practically the only check method used to insure correct reading is the parity check (hardware or by a program). However, this method can be applied to seven- or eight-track tapes only. Some devices have no check possibility whatever. To

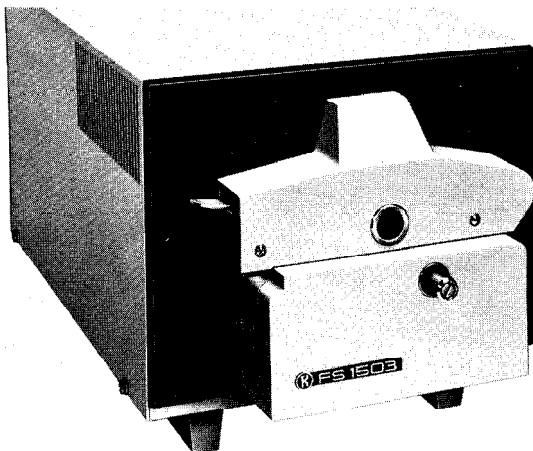


Fig. 4. ZPA FS- 1503 paper tape reader.

vercome this disadvantage, check-sum methods are introduced. These require adding a check row or rows on the tape which, when attached to a group of rows, act as a check symbol, allowing a summation (or hash total) check to be made for that item when the tape is used as input to a computer.

3. Possibility for off-line use of the device.

The so-called paper tape reader-punches are NO separate devices that are mounted under one over.

Paper **Tape Punches**. Paper tape punches (Fig.) as computer output devices are more complex than a simple keyboard-operated punch; their design demands increased accuracy, maximum speed, and reduced maintenance (mainly the sharpness of the punching die).

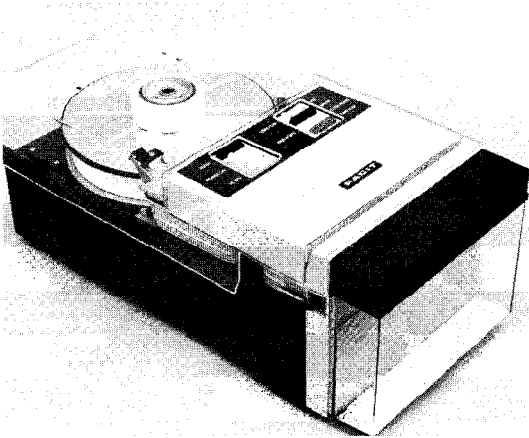


Fig. 5. Facit 4070 paper tape punch.

As computer output devices, tape punches may be classified by their performance. Low-speed tape punches have speeds ranging from some 15 chps up to less than 100 chps, and high-speed tape punches from 100 up to 300 chps.

Some of the more important features and characteristics associated with tape punches are:

1. Automatic winding attachment; same as for tape readers
2. Immediate visibility of the tape after punching, a very useful feature for the user.
3. Number of tracks on tape; five to eight round holes or six-track square holes.
4. Code translation: automatic, matched, programmed, or by subroutine or plugboard.

5. Check control: echo, verify punch activation, read compare, or none.

MAGNETIC-INK AND OPTICAL CHARACTER READERS. Magnetic-ink and optical-character readers interpret information printed or written on a document. The information may be represented in several forms-by marks, bar codes, numerals, or letters of the Roman alphabet, and by other characters.

Marks are made by hand in preprinted positions on the document, each position having its information significance assigned beforehand to express its meaning. For example, a mark can stand for "Yes" in a questionnaire in a particular position, or it can stand for one chosen digit (or number) in the mark field of several preprinted digits (or numbers), etc. Mark readers have long been used with punched-card machines and so-called test-scoring machines. As computer input devices, they are used for many types of applications, mainly for surveys, census compilations, billing, etc.

Bar codes are printed by machine and usually represent numbers selected in a predetermined manner. Bars look somewhat like Morse code representation, but include some type of check. Bar-code readers are used mainly in point-of-sale and similar terminals for reading price tags, identification cards, etc. They are also sometimes used in optical and magnetic-ink recognition systems for subsequent sorting of documents. Numerical digits and alphabetical characters are either printed or written by machine or by hand in a more or less stylized font. This is a steadily expanding field of computer input form in recent years.

Apart from electromechanical scanning, which seems to be less and less used, there are two distinct groups of scanning techniques, magnetic and optical. Both are used at present with magnetic-ink character recognition, a somewhat older technique. However, the commercial production of optical scanners has made distinct progress in recent years. Both magnetic-ink and optical readers are very similar in performance. However, they differ mainly in four ways;

1. The kind of ink used for printing the information to be read by machine.
2. Types of font, the size and the character set they read.
3. Size of documents and volume of printed information on them to be read by the reader.
4. Scanning technique.

INPUT-OUTPUT DEVICES

Table 3. Characteristics of MICR and OCR readers

Characteristics	Readers	
	MICR	OCR
Fonts		
MICR: E-13B	Y	Y
CMC-7	Y	Y
OCR: OCR-A (ANSI I-A or USASCSOCR-A)	N	Y
OCR-B (ISO-B)	N	Y
Other optical fonts	N	Y
Bar codes	N	Y
Handwriting	N	Y
Ink		
Magnetic	Y	Y
Printing (black)	N	Y
Typewriter ribbon (black)	N	Y
Character Density (pitch)		
8 characters/inch in a line	Y	—
10 characters/inch in a line	N	Y
Printed Forms (derived from applications)		
Page: Typical 14 x 9.0 in.	N	Y
Document: 3.75 x 6.0 to 3.67 x 8.75 in.	Y	Y
Journal tape: tally roll:		
1 ft x 1.3 in. to 350 ft x 4.5 in.	N	Y
Readers		
Typical maximum speeds		
Page	N	400–2,400 chps
Document	1,200–2,400 chps	200–3,000 chps
Journal tape (tally roll)	N	1,000–3,600 chps
Sorting possibility (reader/sorter)	Y	Y
Maximum number of lines read per pass	1	*
Error control:		
Validity check	Y	t
Timing check	Y	t
Rescan feature	N	Y

Notes: Y, yes; N, no; —, possible; * up to some 15 on documents and 80 on pages depending on device used.

Consequently, their applications are different. Table 3 shows some characteristics of both types of reader. Magnetic-ink mark recognition (MIMR) readers are being rapidly replaced by optical reading devices.

Magnetic-ink character recognition (MICR) readers interpret only information printed in magnetic ink on one line of a document. The font used may be either the E-13B (adopted as a standard by the American Bankers Association) or the CMC-7 (designed by Bull and adopted as standard font by the European banking community). Both fonts are shown in Figs. 6. A picture of an MICR reader-sorter is shown in Fig. 7.

MICR readers are used mainly in check and credit-card applications. Hence, the document has a small size (typically, 2.75 by 6.00 in. up to 3.67 by 8.75 in.) with one line printed in an MICR font and

containing the numerals and four special characters used for reading-control purposes. Since checks, postal money orders, and credit cards require handling at different points and before their final filing, MICR readers generally have a sorting feature incorporated. Characters to be read magnetically have to be very carefully printed. The reason for using magnetically printed characters on checks is that eventual overprinting by postmarks or smudges will not affect the accuracy of reading. However, some sophisticated OCR methods can also deal with this problem today. Optical readers are described elsewhere in this Encyclopedia.

The preparation of input documents for either magnetic or optical reading should be done with great care. The paper used should be appropriately chosen and print should be clear and well centered. Devices are more or less sensitive to these require-

INPUT-OUTPUT DEVICES

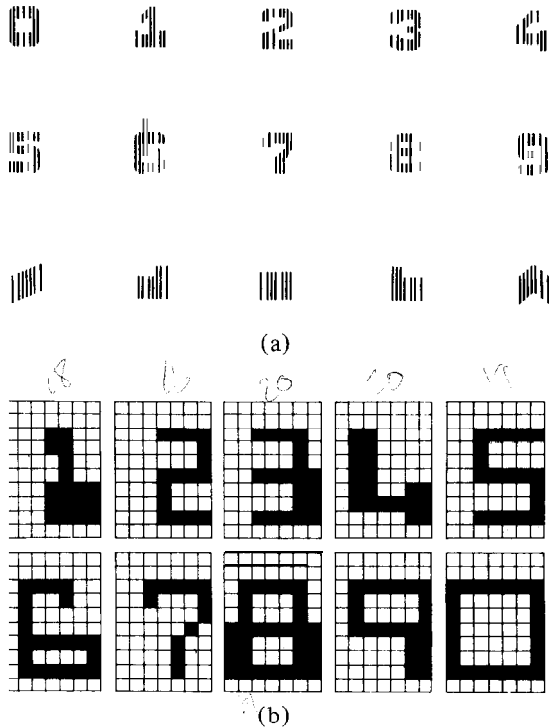


fig. 6. (a) Digits of the CMC-7 MICR font used by the European banking community, and (b) the E-13B font adopted by the American Bankers Association.

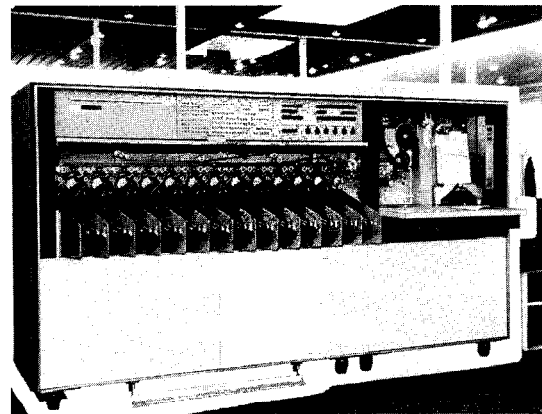


fig. 7. The IBM 1419 magnetic character reader/sorter.

nents. Characters not recognized with a high degree of probability are considered as “unknown” and such documents will usually be marked and/or Ejected in a special stacker called a “reject pocket.”

The handling of rejects, if many, may be very troublesome.

Both categories of readers, MICR and OCR, can be used in an on-line or off-line mode to the main processing computer. When used off-line, the possible output information will generally be written on a magnetic tape. However, punched cards or paper tape are sometimes used as well.

PRINTING DEVICES. Printers are output devices that convert computed data into printed form. The different printing techniques they use have impact upon several features of these devices; these printing techniques are discussed elsewhere in this Encyclopedia.

From the user's point of view, printing devices can be classified into two main groups, those with and without the capability of data input by means of a keyboard. Printers not having this capability can be further divided (by the paper form of the printed output) into strip printers, digital or journal tape printers, serial (character by character) printers, line printers, and page printers. In this article, only the first four categories are discussed; page printers are used mainly in microfilm I/O devices. However, a few page printers using other than microfilm techniques have been included in the later section entitled “line printers.” Note that the term “page” printer is sometimes used to describe the ability of a device to print a page format, as opposed to the “strip” type of printing.

Printers having the keyboard facility have in common (unless directly connected to the computer), besides input and output features, some device to get the connection to the transmission line (e.g., a dial-in telephone). They may be classified into two subgroups, simple and complex devices, depending upon the complexity level of tasks they perform.

Generally, all printers may be used as terminals, but interactive conversational capability is restricted to the keyboard printers.

Strip Printers. As the name suggests, a strip printer prints the information along a narrow (usually half-inch wide) paper tape, much like a ticker tape. It is a low-cost device used for special applications in systems where the cost of a multiple-column printer would be prohibitive. Strip printers are used not only as computer peripherals, but also as telegraph or industrial printers.

A typical example of such a device is the strip printer shown in Fig. 8, which has a printing repertoire of 64 characters as follows: capitals A through Z; numerals 0 through 9; and 28 various signs, symbols, and punctuation marks. These characters are arranged on a print barrel in such a way

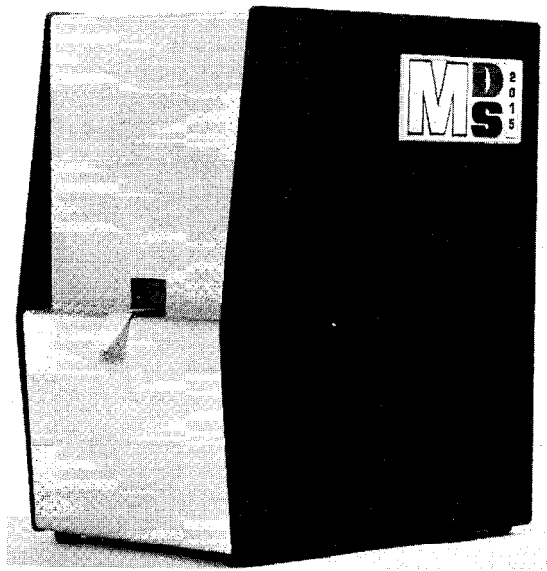


Fig. 8. Mohawk Data Sciences Model 2015 strip printer.

that they pass through the print position in the order used in the ASCII code. Average speed is 20 **chps**; a higher rate is possible for printing numerals only. Characters are printed with their vertical axes perpendicular to the longitudinal axis of the paper strip, 10 characters per inch. The paper stock is a half-inch wide roll approximately 200 ft long.

The usual speed at which strip printers operate is between 10 and 20 characters per second; they can print numerals only, or sometimes also alphabetical characters and special symbols.

Digital (Journal Tape) Printers. The primary advantage of a digital printer is its ability to make a permanent, continuous recording of the numerical values indicated by an instrument over a period of time. A similar requirement is sometimes posed for the output of a low-cost scientific computer. However, for this purpose, an electric typewriter is often used because an alphabetical print-out may be also required in some instances.

The name "digital printer" originates in its industrial application, and is used to distinguish this type of print from the analog one. A digital printer used as a computer peripheral is often called a "line printer." However, as the printed line is very short (between 8 to some 32 printing positions) and the stationery does not generally need to have sprocket holes (which are necessary for the paper-advance mechanism in line printers), the printed tape resembles much more closely that of the journal tape printer.

The speed of digital (journal tape) printers varies considerably, and can be anywhere between 100 lines per minute (lpm) to 2,400 lpm for numerical, or 1,200 lpm for alphanumerical information. The printing set may be either numerical with a few special symbols only, or a full 64-character set. The smaller the character set repertoire, the higher the printing speed usually attained.

Serial Printers. A serial or character-by-character printer is, as its name suggests, a device for serially printing each character, much like a typewriter from which the keyboard has been removed. The printing rate of these devices is usually between 60 and 330 **chps**. The character repertoire usually contains 64, 96, or even 128 characters, often including upper- and lower-case characters and sometimes even a larger "boldface" font, as in the Centronics Model 308 impact matrix printer. The print line usually has 80 to 132 printing positions.

Many of these printers have been designed for use with minicomputers, visible record computers, or as terminals; several are offered also on the **OEM** market. Some may have the optional capability of utilizing a keyboard, in which case they will fall into the category of keyboard printers (discussed in a later section).

Line Printers. Line printers are used mainly to print out results of calculations; they can be programmed to print on stationery preprinted as invoices or statements. The individual pages are part of a continuous sheet and are marked out by folds and perforations across the sheet at intervals required by the nature of the document. The stationery is supplied as a pack, and a complete set may consist of several sheets with interleaved carbon paper to produce additional copies.

The continuous-feed paper supply is fed past the print head by a sprocket mechanism engaged through positionable traction clamps. Vertical spacing and skipping of paper is generally controlled by a tape loop in which the positioning of the page is determined by holes in the specific channels (8- or 12-channel tape being most popular today). When this control is not provided, the program within the central processor must control the line spacing of the **page**.

Fig. 9 shows a vertical-format unit mounted on the left-hand side of the printer. The shaft from the paper-feed clutch extends into the unit to turn a sprocket wheel in correspondence with paper advance. When a format tape is engaged with the sprockets, it is moved between a set of photodiodes and lamps. The tape is prepunched with holes in 12 channels for up to 12 format choices. The holes in

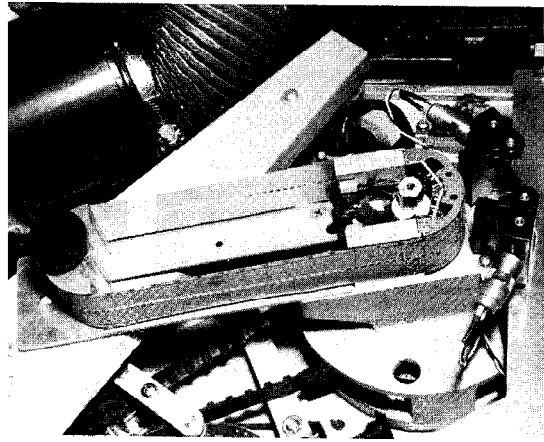


Fig. 9. Vertical format unit for line printer.

Each channel indicate the line of the form at which the skipping paper-feed cycle is to terminate. As the tape moves over the photodiodes, a pulse is generated by each hole. The output from the selected channel is synchronized with the output of the paper-drive pulse generator to signal the end of the cycle. After being printed, the stationery sets can be split up into single sheets by means of decollators and bursters.

Several manufacturers produce computer form printers that copy the continuous output forms from the computer onto single copies of the same or reduced size. Some of these devices can use masks to eliminate copying of certain parts of the forms, or they can add printed information such as headings or footnotes to the printed image.

Some of the more important line-printer characteristics are as follows:

1. Printing speed differs so much that it is useful to divide line printers into three categories. **Low-speed** printers have speeds from about 100 lpm to 300 lpm, with 150 lpm as a good average. **Medium-speed** printers go from more than 200 lpm to less than 1,000 lpm, with an average rate of 600 lpm. **High-speed** line printers go from 1,000 lpm up to more than 2,000 lpm, with an average performance in this category of 1,100 to 1,200 lpm (Fig. 10). All the rates mentioned are valid for a full **alpha-numerical** character set, usually consisting of about 44 characters. However, if a reduced set is used (e.g., **48-character** set or even a numerical set with only a few special characters), often a much higher rate can be attained, sometimes up to double the rate of the full set. The possibility of using more than one character set for printing depends upon the printing



Fig. 10. Potter LP 3403 (1240 lpm) chain printer.

technique used. Printers attaining much higher speeds than those mentioned here are also available.

2. The maximum number of printed characters on a line is usually 120, 132, 136, or 160, but any number between 72 and 200 is possible.

3. The print density on a line is usually given in "chpi," the number of characters printed in 1 in. (also called "character pitch"). It is generally 10 chpi; however, MICR font printers use 8 chpi, and other exceptions are also possible.

4. Start-stop or continuous operation is available. Generally, all line printers operate in the start-stop mode. **However**, printers using certain printing techniques require continuous operation and are used for off-line batch processing.

5. The printing facilities of line printers generally employ the mechanical printing barrel (drum) or the chain (train) techniques.

6. Image printing is not possible with line printers. However, some simple graphs or roughly drawn curved graphs, using the character set available and the overprint or overlay technique, can sometimes be printed (see later section, "Alpha-numerical Visual Displays").

7. Checking involves parity, timing, echo, validity, receipt of data, or none.

Other Printing Facilities. One field of computer printing application, which is relatively new, is

“computer-aided correspondence.” The principle of this application is based upon the so-called text-tins, which are short text sentences or articles stored in the computer. Writing of individual letters is done in a batch processing mode of operation (e.g., once a day). Input information for each letter to be written contains identifiers of the “text-tins” to be selected, together with the necessary variables to be inserted into the selected text (such as date, rate, etc.). An optical mark recognition (OMR) reader may be used as an input device for this kind of information. In some applications, as in the banking and security industries, additional information from the computer data base can be added to the text. All this information is then processed on the computer and the required letters are printed.

Computers are used also to prepare magnetic tapes (or paper tapes) to control typesetting machines in the printing industry. The processing generally proceeds like this: The original text is written on a magnetic tape and printed for proof-reading. All corrections and amendments resulting from this proofreading are written on a second tape, and both tapes are again processed on the computer. The final output is a magnetic tape containing the corrected text and commands for a typesetting machine. Its operation is subsequently automatically controlled.

In a similar application a computer is used to generate Braille prints. With the aid of translation tables, the computer translates the text to be printed onto some medium suitable for controlling the matrix disk of the Braille dot typesetting device used. Similar systems using Telex and Braillemboss terminals connected with a central computer (used for translation) are under development. Another solution to Braille printing problems has been reported by a manufacturer who supplied a computer system to the Southern College of Business in Orlando, Florida. The system uses an eight-line per inch printer, and a special conversion program changes the alphanumeric print-out into the Braille system of dots. The impressions in the paper are formed by increasing the force of the print hammers, which strike through a specially designed sheet of thin felt, causing the effect of “exploding” the dots.

Line printers are used also as devices in computer graphics (CG) to produce, for example, pictorial data mapping. A software house in Great Britain has developed the LINMAP system, available commercially, which produces statistical maps on a line printer, using ten symbols to simulate the degree of shading density. There is very little restriction on the format of the input data, and up to 400

data items can be handled for each coordinate point. The same firm also offers an additional service, CILMAP, which produces art-printed color maps. This is done by transferring the LINMAP output to magnetic tape and then employing a photoelectronic typesetter.

The Canadian government uses computers to generate a series of alternate land-use maps. The computer output is done in a color-coded system analogous to standard land-use schemata. The scaling technique used makes it possible to select one individual unit of low-density housing and then expand this unit to the size of the entire sheet. Then the program can be used to plan the individual residential unit. Thus, for each individual unit, a series of matrices based on various criteria can be devised to formulate the living patterns of the clients.

A generalized spatial allocation procedure, called “ALOKAT,” has been applied to various problems, including individual houses; a neighborhood project; an intensive-care ward of a hospital; a student union in Syracuse; a car manufacturing plant in Algiers; a faculty of law in Marseille; a new university with teaching, research, and laboratory units in Bologna; an office building in Philadelphia; and a simulated town. In all cases except the town (for which a plotter output was used), chain printer output (a special 8 by 10 instead of the standard 6 by 10 chain) was used as the least expensive, most readily available means of producing graphic output. Overprinting of characters was used to give solid borders to the area.

Imperial Chemical Industries, Ltd., England, has developed a set of software packages called “CROSSBOW” for applications in the chemical industry. The set consists of four programs intended for use by universities and research institutes all over the world, since the formulas of chemical compounds are an international language of chemists. One of these programs, for example, is able to answer queries concerning a given compound—such as whether it is original or what other chemical compounds it contains—and directly prints its formula in the conventional way.

Facsimile line printers using dot-printing techniques are used for data and image transmission. However, the number and density of dots is higher than in the conventional line printers, and the spacing between two dots in the line is always the same.

Simple Keyboard-Printers. In the subgroup of relatively simple devices, there is always an input keyboard and a printer. The latter can often be used to print both the keyed-in data and the computer

output information. In this subgroup are teletype-writers as well as typewriters and typewriterlike devices or printers of some other type with a keyboard added. These devices are often called "keyboard printers."

The input speed of keyboard printers is limited by human ability factor; their output speed is determined by the device and/or transmission capabilities, ranging between 10 and 40 chps for interactive terminals, but up to 180 chps for some terminals. The printing technique used is usually of a serial type, with character-by-character print.

The arrangement of the input keys may be that of a separate keyboard for alphanumerical information-upper and/or lower case-and/or for numerical information entry. Apart from these, some controlling keys are needed; they may be accommodated on the same panel with the others or on a separate keyboard.

The interrogating function is sometimes provided by devices called "interrogating typewriters." A single unit may be connected directly to the processor (generally through an interface channel), or a number of devices may be connected via a communications multiplexer. Besides the interrogating function, the device may be used also for other purposes, such as for program debugging.

This subgroup also includes portable terminals, which incorporate a built-in telephone coupler and typewriterlike input and output features.

More Complex Keyboard Printers. This subgroup includes devices of the types mentioned earlier as simple keyboard printers (with the exception of portable terminals), but with some more features added to them, such as reading and/or punching of paper tape. These devices may have some kind of programming feature that allows for a restricted computing facility and printing format, and often for a choice of a few available programs (Fig. 11). These features allow for less complex programming at the computer site and considerably lessen the number of commands transmitted from the computer to the terminal. The programming feature may be external [e.g., plugboard or programming bar] or internal [hardware and/or software type]. Devices with more sophisticated (computerlike) facilities are often called "intelligent." Similar "intelligent" terminals are also found in the visual display unit (VDU) group or among graphical devices. There may be systems configurations making use of all these devices.

DIRECT KEYING DEVICES. Direct keying devices represent a relatively new group of computer input equipment, enabling direct entry of information by



Fig. 11. Olivetti TC 300 keyboard-printer terminal with paper tape reader and punch.

means of keyboards (or, exceptionally, dials) operated by humans. Many of these devices are similar to those used in data collection systems that operate in an off-line mode, mainly to keying units in key-to-tape or key-to-disk systems. Direct keying devices, however, are connected with the computer directly, either by cable or transmission lines. Thus, they can also use for validation purposes the files stored in the computer direct access memory. Given proper environmental conditions, they can be used in time-sharing systems that require several types of intermixed data formats to be processed in real-time mode (e.g., updating centralized computer files).

Keyboard devices are used for entry of variable data. However, some of them may have additional features that permit duplication of repetitive data or of personal or other identification, using prepunched or preprinted cards, badges, edge-punched cards, etc.

To allow the operator to correct a typing error detected during the typing operation, the keyboard device is often connected by either some printing means or a simple visual display called "key-input checking VDU." (See the following section, "Alphanumerical Visual Displays.")

As do all devices that use a keyboard, direct keying devices are faced with the problem of the keyboard arrangement, which must be adapted to generation of the character repertoire of the ISO seven-bit ASCII code.

INPUT-OUTPUT DEVICES

Direct keying devices can be divided in four categories; numerical keyboard devices, Touchtone telephone, alphanumerical keyboard devices, and special keyboard devices. Each of these is discussed in subsequent paragraphs.

Numerical Keyboard Devices. These devices resemble those used in key-to-disk off-line systems, but they differ in several respects. The keyboard for a numerical keyboard is arranged in three basic configurations, as follows:

1. Adding machine

7	8	9
4	5	6
1	2	3
0		

2. Punched-card machine

		0
1	2	3
4	5	6
7	8	9

3. Telephone

1	2	3
4	5	6
7	8	9
0		

Differences between these basic types of numerical keyboard arrangements are self-explanatory, as are their uses. A few functional keys are generally added to the ten numerals.

Touchtone Phone. The advent of the electronic telephone exchange in the United States made possible the Touchtone phone, which provides a telephone-to-tape data-entry method. The instant response of the new exchange relays eliminates the requirement for a spring-loaded dial as a counter. Instead, the relays respond to unique tones generated by keys that replace the dial on the telephone. Soon after the introduction of Touchtone, it was realized that the instrument provided a new means of entering or preparing data for a computer, particularly where the data has to be collected from a number of remote locations, as in a multi-outlet or branch type of business, or from many different departments situated at some distance from the data processing center.

The Touchtone phone is a special kind of numeric keyboard device. When used in **telephone-**

to-tape communication, it makes use of a translator, an electric device that interprets the unique tones from each phone key, alters them into a computer code, and enters them directly onto tape. The tape and translator are connected to an automatically answered telephone (data set), which allows many **phones** to input through a single translator onto a single tape. A data set is a device that connects a data processing machine to a telephone or telegraph communication line. A telephone data set is a unit used to connect a data terminal to a telephone circuit; e.g., to transmit data from the terminal to the processing center. Such a data set converts signals from the terminal into a form suitable for transmission over a telephone circuit, and vice versa.

Typical Touchtone telephone users key in about 1.4 digits per second, with a relatively low error rate. Errors of omission are the most serious faults to watch for. Where check-digit verification can be applied, the error rate will be even lower. Further savings may be achieved by the use of a plastic card (originally designed as a self-dialing facility) for entering fixed format information.

The technique of Touchtone data transmission is not limited to batch operations, but can be used on line for any variety of information updates applicable to production, stock, and credit control systems. Increasing use of credit cards opens up a new area of on-line credit validation for all sorts of businesses by keying on line to a central computer. In countries where the electronic exchange is not yet standard, this device can still be used, once a connection has been established through the normal dial telephone. Touchtone pads, linked to any telephone and transmitting through the data set and translator, allow remote data preparation direct to tape.

The Touchtone pad has a single **12-button** keyboard and is easy to use. The pad provides the user with the normal ten keys of the Touchtone phone, plus two additional keys (one for "skip," "duplicate," and "data entry" and the other for "error correction"). Data is entered a line at a time; each line can contain as many as 180 characters and can be split into as many fields as required. One key is used to skip from one field to the next; at the end of the line, a final tap on the "skip" key causes the line to be written onto the key tape. If an error is made during the entry of the line, the error-correct key can be used and the line reentered before it is written to tape.

Alphanumerical Keyboard Devices. Alphanumerical keyboard devices are similar to the numerical ones previously described, the only difference being the keyboard, which may be one of three

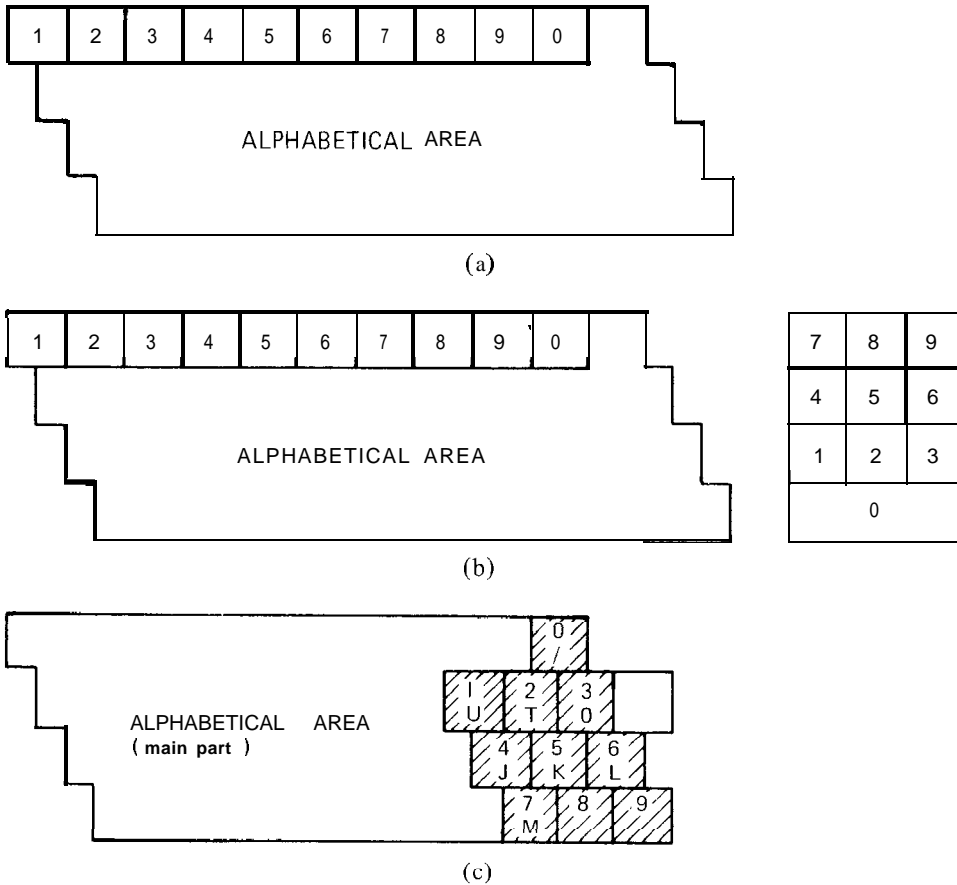


Fig. 12 The three basic types of alphanumeric keyboard arrangement. (a) Typewriter-type alphanumeric keyboard arrangement, for predominantly alphabetical data. (b) Typewriter-like alphanumeric keyboard arrangement, for predominantly numerical data. (c) Card punch-like alphanumeric keyboard arrangement. This type allows numerals and some special characters to be written in large size (like capital letters), the alphabetical characters being typed in lower case (small letters). Usually, two shift buttons are provided, one for switching from upper to lower case and the other for switching from lower to upper case. If used for printing also, the alphabetical characters would usually be capital letters.

basic types, depending upon the positioning of the numerical characters in the overall layout of the keyboard. A typical layout is shown in Fig. 12.

Special Keyboard Devices. Special keyboard devices are of two types. In one, a normal numerical or alphanumeric keyboard is used, but keys are assigned special significance, depending upon the type of work for which the device is being used. The keys may be given special additional labels that identify various types of information when the recording is made. When the recorded data is read

into a computer, the computer program is so devised that the significance of the various items of information is recognized.

The second type of special keyboard devices is that especially designed for a given purpose, such as mathematical expressions. This special keyboard device is used mainly in addition to keyboard devices of the numerical or alphanumeric type.

ALPHANUMERIC-AL VISUAL DISPLAYS. A display device allows the operator or user to visually inspect data that is keyed into the computer, and/or re-

INPUT-OUTPUT DEVICES

trieved from the computer upon a request from the operator/user, or displayed automatically as a message. For example, data may be presented as a printed report, or in graphical or character form on a cathode-ray tube visual display unit (VDU). This section will describe the alphanumeric VDU, or alphascope. (See later section, "Graphical Visual Displays," for a discussion of graphoscopes.)

Key Input-Checking VDU. To ease the key-in operation, several kinds of visual aids are used, such as a simple illuminated panel display or the more sophisticated alphascope with cursors.

On an easy-to-read illuminated display panel, the operator is shown the current mode (such as **WRITE, VERIFY, READ, PROGRAM, SELECT, OR PROGRAM ENTRY**) and status (such as automatic skip/duplicate or error conditions) of the key station, the field the operator is keying (in standard English text), together with the last character keyed.

The more sophisticated key input-checking devices incorporate a VDU upon which the latest data entry or any record entry held within the control unit buffer storage can be displayed. Such devices usually use a **64-character** set. Normally, the data is displayed in a few lines, the total number of characters being limited by the maximum number permissible for a single record (generally less than 200 characters). Often, there is also a moving cursor to show where the next character is to be entered.

All these devices have in common the capability to display the input data only. If a device has the additional capability of displaying the output data also, which is the more general practice, it is classified in a separate subgroup (see next section, where some common features will be discussed).

Alphascope-The VDU as an I/O Device. An alphascope is an interactive alphanumeric device (often a terminal) that forms part of a computer-based system requiring a short response time for getting answers to queries made by managers, dispatchers, stockkeepers, or clerks. Its purpose is to retrieve these answers from the computer random access memory. The alphascope terminals need only be connected to the computer by an ordinary telephone line. The cost of each terminal unit is about \$4,000. Since it is a relatively low-cost unit, it makes possible later extension of the user's facility to true computer graphics, from simple graphs to full vector capability.

The CRT (cathode-ray tube) terminal is not much more complicated than the Teletype, but because it does not depend on a mechanical means of printing, it is considerably more reliable.

An alphascope consists of the CRT, a keyboard, a method of generating characters, a method of refreshing the display, and communications equipment. The *keyboard* fulfills the data input function and is generally arranged like a typewriter with a few more control keys added. Some keyboards have a lower-case alphabet in addition to the upper case, and entry of mathematical and other special symbols is often possible (Fig. 13). The keyboard also controls the screen location of a cursor, which is a movable dash symbol that glows beneath the position at which the next character from the *keyboard* will be displayed. Often an optional feature is available to allow the display stations to have full editing capability. This permits the display operator to insert and delete characters, move lines up and down, set tabs, and provide operator status to the CPU.



Fig. 13. The UNIVAC Uniscope 300 visual display terminal.

Screens have generally different diameters, to 20 in. or more, with the display on the whole screen or on part of it. Information is presented from 6 to 20 lines, with up to some 80 positions each, giving a total capacity from 250 to a maximum of some 2,000 characters. The characters are usually displayed in fixed positions on the screen, with the beam moving along each row, character position by character position. More sophisticated systems switch the beam to the next row when no more characters in the row are to be displayed, thus eliminating the need for tracing the path of remaining blank positions in the row. Sometimes a roll feature selects the whole array of characters to be displayed, rolling either in line-by-line or in group-of-lines mode.

Another type of input is a *touchwire* system. Ten or fifteen short pieces of wire are imbedded in a transparent screen over the face of the display tube. The function of each wire is displayed on the screen above it so that it is possible for every character on the keyboard to be displayed by an associated touch, thus eliminating the need for a keyboard. The wire is connected to a balanced electrical bridge and is sensitive to the touch of the operator. By putting his finger on one of the short pieces of wire, the user initiates the start of a computer program.

The user is piloted through the process he controls by labels alongside each touchwire, which are initiated on the display by the computer program. He looks at the displayed diagram (say, an electronic circuit) and points to parts he wishes to alter. Touchwire systems are presently used only to choose the next required display, since a balanced-bridge network is more expensive than a keyboard, and also does not satisfy the sense of touch needed by typists.

For drawing of the required character, a *hardware character generator* is used*. Of the many types applied, the most common defines a character by brightening the required dots on a matrix (typically, 7 by 5). Some techniques "paint" lines between points on the matrix (Fig. 14). Very fine characters are produced by a monoscope system, which requires a second tube on which the character set is etched.

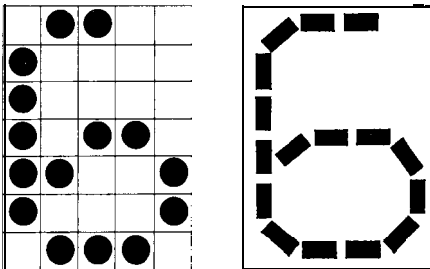


Fig. 14. Dot matrix and painted-line matrix presentation on the IBM 2260 and 2265 VDU, respectively.

For keeping the text displayed, the method of *continuous cyclic refreshment* of the picture (40 to 60 times per second) is generally used. Several problems are connected with storing the picture. The delay lines currently used have the additional problem of temperature control. Use of a core store is almost prohibitive because of its cost. Storage tubes would solve the problem, but they are expensive and cannot

be selectively erased (which is a very useful feature required by the user).

The *communications* with the main computer should be much faster than that of a Teletype if the main advantage of the alphascope (the quick presentation of information) is not to be lost. The reading speed should be much higher than that of the human eye. This is essential in many applications because the user scans the displayed text for quick orientation, choosing only the parts required for more detailed study.

Simpler display techniques may be sometimes adequate, depending on functions they have to perform. Fig. 15 shows a display designed to accept input data keyed into a buffer store as well as output retrieved from the internal memory of the computer. However, only numerical data can be displayed on this screen.

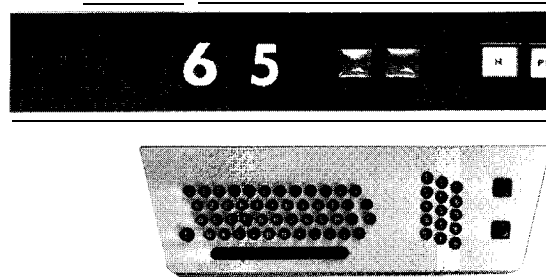


Fig. 15. Six-character numerical illuminated panel display of Honeywell Bull Gamma 55.

Plasma and Other Output VDUs. Other hardware solutions are sometimes considered for visual display output devices, besides CRTs. One of the latest is the plasma display (Fig. 16), consisting of three sheets of glass. The middle sheet is drilled with holes 0.025 in. apart, each containing receptacles filled with illuminant gas (plasma). Both outer glass sheets incorporate strips of transparent conductors, one vertical on the inside sheet and the other horizontal on the outside sheet. The a-c voltage is kept at a level required to make these tiny discharge tubes glow when fired at the user's control station, and any particular spot can be switched on or off. The display provides its own storage, and there is no limit to the size of the screen.

AUDIO DEVICES

Audio Input Devices. Audio input devices are still in a research and development stage, although a few systems are commercially available. Automatic recognition of the human voice is extremely difficult,



Fig. 16. CDC plasma display at New York Stock Exchange.

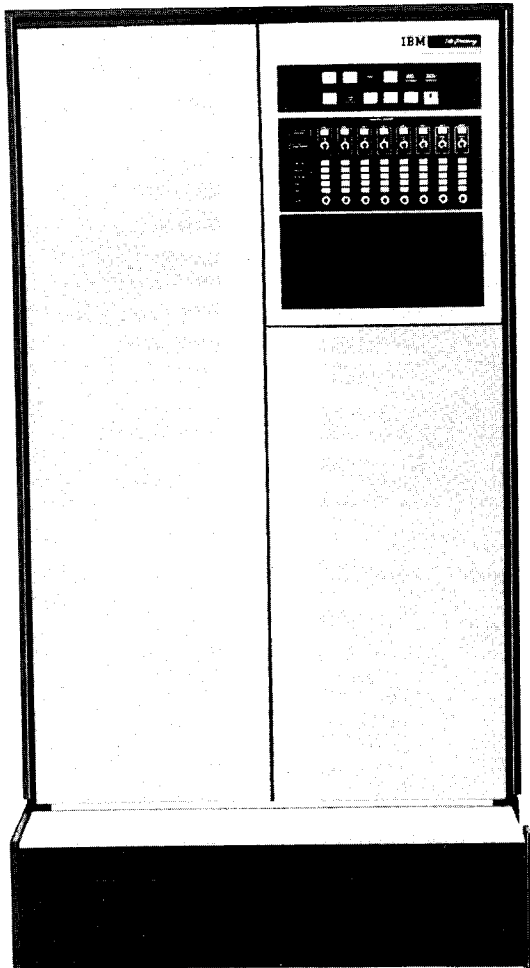


Fig. 17. The IBM 7770 audio response unit.

mainly due to the segmentation problem (i.e., the recognition of boundaries between spoken words) and to the extremely wide differences in enunciation among people. However, the recent electronic development of recognizable *voice prints* makes it possible to recognize a person by his mode of speech. At present the use of voice prints is limited, but they may introduce new methods into banking and other money-transfer computer applications, especially in the area of program security. One company, Threshold Technology, has produced some voice recognition systems that can compare a speaker's voice with stored voice patterns. One such system has been attached to a Data General Nova minicomputer to help route baggage and perform other tasks in the United Airlines terminal at O'Hare International Airport in Chicago.

Some computer manufacturers have produced input devices that recognize a few English words (10, 16, etc.) to be used as computer commands, but these devices need much improvement before they can become generally useful.

Audio Output Devices. Audio output devices are produced by only a few manufacturers. Figs. 17 and 18 present examples of two different approaches to their design. Both units provide a recorded voice response (optionally, male or female) and sometimes with language options to inquiries made from telephone-type terminals (with dials or Touchtone phone keyboards) and similar computer transmission terminals (with keyboards). They are attached to the computer via the multiplexer channel that connects the computer to the telephone network. The main differences between the two units are as follows:

1. The first audio response unit (Fig. 17) provides a vocabulary prerecorded in a digitally coded voice on an external disk file. This offers an unlimited vocabulary with a small number of lines (two basic lines, expandable to eight).
2. The second audio response unit (Fig. 18) provides a vocabulary prerecorded in analog form on a magnetic drum within the device. This offers a maximum vocabulary of 128 words with many lines (4 basic, expandable to 48).

Devices of the second type are used, for example, to answer calls to out-of-service telephone numbers. An operator asks the caller for the number he wishes to reach and connects it into the system through a simple ten-button keyboard at her station. The computer searches the directory file (stored on a disk) for the new number and then transmits it to the

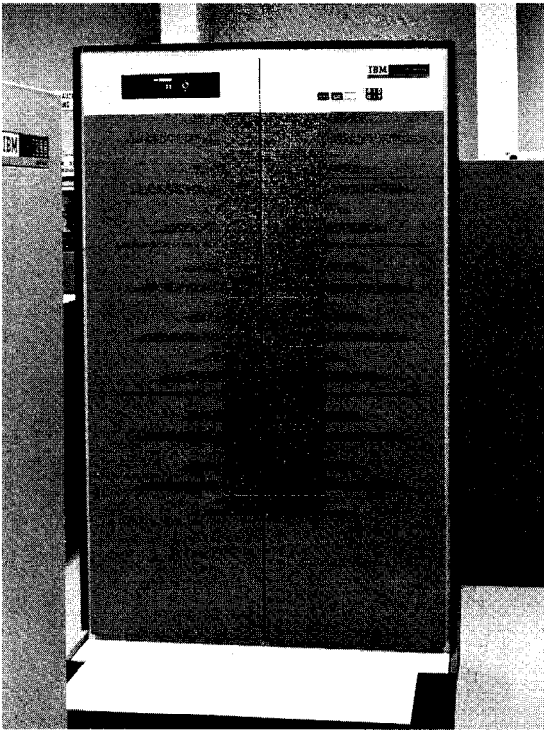


Fig. 18. The IBM 7772 audio response unit.

audio response unit. A voice message is heard by the caller within 10 to 20 sec after placing his call. Devices of the first type are used in more complex systems, but the number of transmission lines is limited.

I/O DEVICES IN CONTROLLED PROCESSES. Processing industries have grown up with analog presentation of data. Digital machines are now coming into this field, some in an indirect capacity-supervising analog controllers-and some observing and optimizing processes. In recent years, with the development of digital computers, digital machines are beginning to take over a very substantial part of process control tasks, often together with some analog elements if necessary, thus forming a new technical field of applications called "DDC" (direct digital control).

There are two main forms of DDC. In one, local control loops take care of subordinate functions and the central processor exercises supervisory control over local loops. In the other form, a large-scale computer sends signals to control devices directly, with virtually all controlling functions being exercised by the computer.

Input Devices for Processing Industries. Computer input in process control systems involves the following types of information:

1. Descriptions of type, quality and quantity of the incoming resources to the process (e.g., raw material and energy).
2. Description of the transformation process that takes part in the plant (e.g., process variables such as temperature, pressure of fluids, flow of fluids).
3. Information signaling interrupts of the process and describing their causes (such as equipment breakdown or some other abnormal condition), as well as commands for measures to be taken, which are given automatically by the computer or by the supervising engineer.
4. End-point control information describing type, quality, and quantity of the process output (e.g., finished products, materials and energy not consumed in the process).

With few exceptions, the input information is procured by suitable industrial sensing devices that form the first link in a chain of control equipment. Some of them substitute for human senses:

Feeling: temperature or thickness.

Sight: light, color, smoke density, level of liquids, or granulated solids in open containers, dimensions of solid bodies, etc.

Hearing: human speech recognition.

Taste: acidity, salinity, sweetness, etc.

Smell: presence of odoriferous gases such as ammonia and coal gas,

Some sensing devices have "extrasensory" characteristics beyond direct human perception, such as moisture measurement, chemical analysis, and crack detection.

Before a variable of an industrial process can be controlled, it must first be measured. There are two ways to achieve such measurement. In the first, some physical property of a sensing device can be utilized directly (e.g., a spring, balance, thermometer, barometer, or tachometer). In the second way, comparisons have to be made with a known but adjustable quantity of the same nature, in which the process of measurement involves the accurate assessment of equality between the two qualities (like a scale balance, measuring rule, micrometer, and potentiometer).

The sensing and the measuring functions are usually combined in one instrument. As processes

INPUT-OUTPUT DEVICES

get more complex, both the number and type of variables that need to be controlled increase. Information from a multitude of instruments has to be concentrated and transformed into a standardized digital format that is suitable input information to the computer.

The forms of data presentation from the various sources differ considerably and their frequency can also be widely different. The process variables measured by industrial instruments have to be filtered and amplified, switched, or scanned by multiplexers, which sequentially connect these instruments via analog-to-digital converters to the central computer buffers. Other forms of data presentation may be pulse inputs that enter the buffers via counters, or binary inputs and process interrupts that are connected directly to the buffers.

Output Devices for Processing Industries. The computer output has two distinct functions. The first function is the automatic regulation of the process. The output electric signals set up and/or adjust the electric actuators of the regulating devices (valves, etc.), thus forming, together with the automated input, a closed-loop control system. This type of output usually is in the form of electric signals that have to be converted from digital to analog form and then amplified; there may also be a binary or a pulse output that can be used without any intermediate device whatsoever.

As opposed to the machine/machine interface function, the second function of the computer output concerns the man/machine interface. This normally gives up-to-the-minute information to the supervising engineer, usually signaling some abnormal condition or possibly calling for his takeover of control from the computer. It also gives logging information on the process (for further analysis of behavior under varying conditions), using typewriter, Teletype, printer, or plotter.

SPECIAL I/O DEVICES. From time to time input/output devices designed for some special application can be used. Two examples of such devices are discussed here.

Ticket Vendors. Early in 1970, two experimental passenger-operated automatic ticket vendors linked to a computer were installed at Chicago's O'Hare International Airport. The ticket vendors accept magnetically encoded credit cards issued by American Express and American Airlines to a selected group of passengers who frequently fly out of Chicago. The system is intended to test whether self-service machines can speed passengers through the airport and how travel agents can benefit from automatic devices.

A passenger with an advance reservation inserts his credit card into the vendor and presses a "Yes" button in reply to whether he has a reservation. He then removes his credit card, and his ticket is delivered within a minute. Without a reservation, a passenger may reserve a first-class flight or coach accommodation on the next available departure to 11 possible destinations.

The automatic ticket vendor consists of a display panel, a credit-card reader, and a ticket issuer. The display panel contains the instruction messages, pushbuttons, and digital display tubes. The display tube assemblies show available flight departure times, controlled by a message from the computer to insure that the correct time appears. The credit card is inserted into the reader while the data recorded on the magnetic track is processed. Data includes the card holder's name, account number, issuer code, and also the qualifier codes used for billing and to check whether the card has been lost or stolen.

Tickets are issued as one or more flight coupons bonded together in a book form with a passenger receipt coupon. Data is written onto the magnetic track across the back of each flight coupon and is then checked. After a ticket has been removed, an auditor's coupon is prepared and stored in the ticket vendor.

The system has developed as a result of problems involved in coping with the expected 300 million air passengers a year in the United States. With the introduction of high-capacity jets, the passenger flow problem at airports is becoming increasingly serious. Similar devices have been designed to form part of computer-controlled issuing systems for betting tickets, and are in operation in France, Australia, and the United States. Figs. 19 and 20 show one such system installed with the Totalizer Agency Board, Melbourne, Australia, and the special telephone display units used at the telephone "betting auditorium."

Some currently offered devices are the

1. Vogue Model 8 10 airline ticket printer, manufactured by Vogue Instrument Corporation of New York.
2. Di-An Series 8000 from Di-An Controls of Boston.
3. CDC Remote Ticket Issuing Machine (TIM) manufactured by Control Data Corporation.

These devices are used in various systems: besides airline seat reservation and ticket issuing, they act as totalizers in subway systems and race

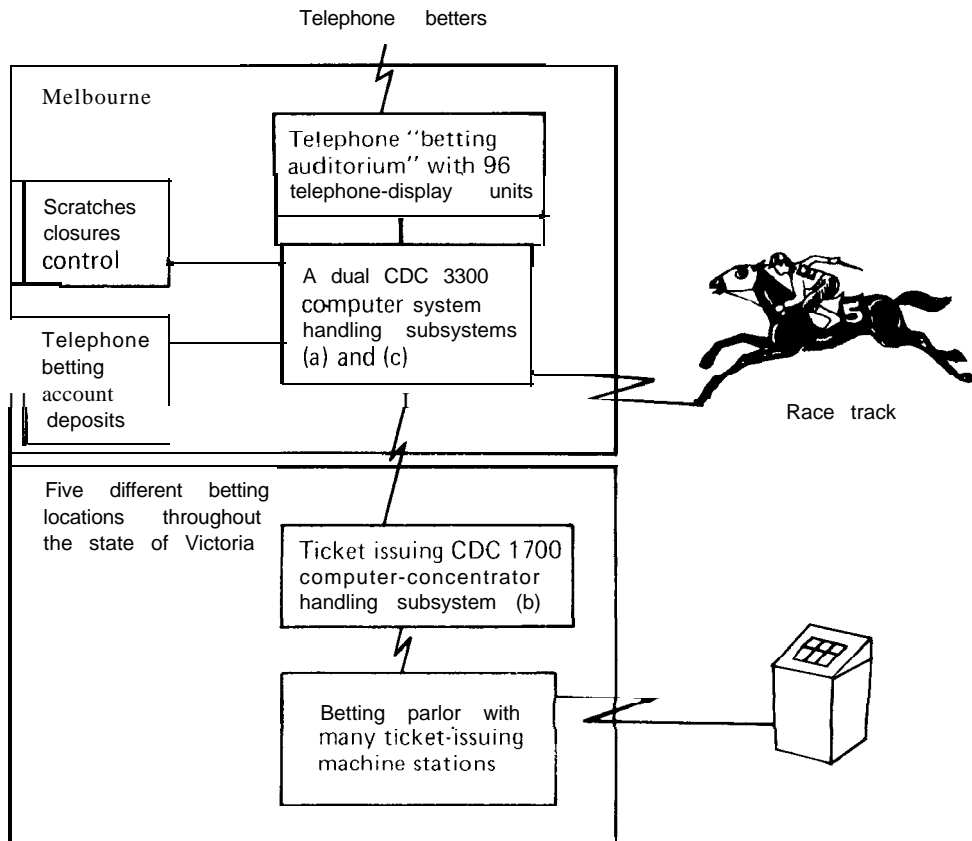


Fig. 19. Sketch of the CDC 3300/1700 system used by the Totalizator Agency Board, Melbourne, Australia. Subsystem (a) at the Melbourne Board handles bets placed by telephone only. Subsystem (b) at the betting stations record bets made in person, issues tickets to bettors, and pays dividends or winnings. Subsystem (c) at the Melbourne Board finalizes pools, determines dividends, processes miscellaneous business data, and keeps records.

tracks (pari-mutuel), and in sports and theater-seat reservation systems.

Cash Dispensers. Cash dispensers, sometimes called "cashpoints" or "self-service banking terminals," (Fig. 21) are computer terminals normally installed in a banking environment or in shopping centers. These terminals allow customers of the bank to withdraw money up to a certain amount. When linked directly to a computer, they extend real-time credit authorization.

Cash dispensers were first introduced in the early 1970s. The main manufacturers were originally Burroughs and IBM. They were equipped with a keyboard and a slot for insertion of a credit card. Depending upon the model used, the transaction took between 10 and 40 sec.

In the Burroughs models, protection against forgery was by means of a secret identification code stored, along with other data, on a three-track magnetic strip on the credit card. In the IBM models, each customer was given a code number, separate from the credit card, which had to be keyed in and which was checked against the same number stored in computer memory. This last method also provided protection against the misuse of lost or stolen cards. The Burroughs system allowed a similar check in which the customer keyed in an account number that was also registered on the magnetic strip. In all systems the keyboard was used for specification of the amount of money required by the customer.

Currently, other manufacturers are also producing cash dispensers, often with additional func-



Fig. 20. CDC telephone display unit designed for Australian TAB system.



Fig. 21. The IBM 3614 self-service banking terminal, a cash dispenser.

tions such as charging a withdrawal to a credit card account, making deposits, or paying bills. Sometimes, as in the case of Docutel's Total Teller, a receipt is returned to the customer. These more complex systems usually allow customers to perform transactions at any time of the day. They use display panels to guide the user through the specific transaction. Such automatic teller systems are often built around a minicomputer, which may operate off line or on line to the bank's main computer. If, in the

latter case, there should be any breakdown in communication, then the system switches immediately to off-line operation.

Graphical Input-Output Devices. The term "computer graphics" (CG) is used to denote a set of computer techniques and applications wherein data is either presented or accepted by the computer in the form of line drawings or graphs. The interest in CG and new developments in hardware, software, and their applications has grown steadily since the mid-1960s, mainly after the introduction of interactive graphic devices, time-sharing systems using terminals, and the possibilities of plotting projections of three-dimensional objects. Computer graphics and image and picture processing are discussed in more detail elsewhere in this Encyclopedia. This article is concerned only with the CG I/O devices.

CG input/output devices can be classified into four main groups:

1. Image input devices (derived from hard copy graphics).
2. Electromechanical plotters (such as output devices onto hard copy).
3. Video display units (VDU) with graphical capabilities (such as interactive graphics devices).
4. Microfilm I/O devices.

Nearly all devices of these four types, in addition to their capability to represent graphical images, also allow alphanumerical characters to be represented. These more sophisticated graphical devices are much more expensive than the alphanumerical ones, and the success of their application may be seriously affected by unsuitable or unavailable software.

IMAGE INPUT DEVICES. Many applications require transformation of hard copy images into a proper form for computer processing. Methods used for such computer input are closely connected with those used for facsimile data transmission devices, even when these are not used as computer peripherals. However, some of the scanning methods used and their envisaged incorporation into computer-based systems (like banking, credit card, or message switching systems) make them worth mentioning here.

Facsimile I/O Devices. Facsimile data scanners often use some optical raster-scanning method that optically scans all points of an imaginary grid placed over the picture to be transmitted, registering each point as black (part of an image present) or white (part of an image absent). In this technique, even if

only a minute portion of the picture is actually filled by an image (e.g., a signature), it is then necessary to transmit details of all the background that carries no useful information. This basic technique can be improved, of course, by application of some methods to eliminate this drawback, improve the shading, introduce colors, etc.

In all facsimile transmission systems the original hard copy picture (or a portion of it) is sent over to the terminal (or terminals) where it is printed or projected onto a screen. The projected image can be microfilmed at the terminal by a camera so that a hard copy representation of the original image can be obtained at some later date. However, in facsimile transmission systems, data is not represented in a digital form.

For graphical processing on computers, the information must be recorded in a digital form, either directly or with the use of a converter, when an analog form is to be translated. In addition to techniques in which the pictures are usually sensed and reproduced line by line, there is another method by which the images are drawn by incremental plotting. Here the information is represented by scales, coordinates, vectors, etc., as well as by identifiers, some descriptive text, and figures, etc. In some parts of the process, where it is necessary to translate the information from one representation to another suitable for computer processing, the computer itself can help substantially, thus reducing the usually great burden connected with manual description of graphical information in the required digital form. Besides the computer itself, the user has a wide variety of devices and means designed to help him in the description task; they can be classified as follows:

1. Devices designed to help the user with the manual description and/or registration of the image data, to be used for subsequent computer processing.
2. Same as the preceding class, but for use in a direct interactive on-line operation mode.
3. Devices used for automatic input of image data to the computer.

Because of the wide variety of devices designed for different applications, only some representative examples of devices in each group will be given here.

Class 1, Devices designed to help the user with the manual description and/or registration of the image data, to be used for subsequent computer processing. One of useful devices in this group is the *pencil follower*, which is used for conversion of data pre-

sented as graphs, charts, drawings, photographs, and film into digital form for subsequent automatic processing by a computer. It generally consists of two units, the reading table and the electronic console. The pictorial information to be analyzed is placed or projected onto the surface of the reading table. Operation is effected manually by following the trace with the reading pencil or by pointing the pencil at a position. An automatic mechanism beneath the table surface follows the pencil accurately, and position signals are passed to the electronic console where they are visually displayed and converted into suitable form for feeding the output devices (punched cards or tapes, a typewriter, etc.).

Operation of the pencil follower is normally controlled by a foot switch connected to the reading table. Alternatives are a handheld pushbutton; in some cases, a pushbutton is incorporated with the reading pencil.

Usually there are no lines engraved on the reading table for alignment of charts, graphs, or drawings, but such lines may be added by the operator if required. The normal method of reading is to place the graph or chart at any position or angle on the reading table and to take off fiducial points prior to the main analysis, programming the computer to take care of any required correction. Another method requiring special types of pencil followers is the reading of images such as films, high-speed camera films, and other types of photographic work projected on the reading table.

The speed is normally between 18 to 300 symbols a second or between 2 and 30 pairs of coordinates a second, depending upon the ability of the operator and the speed of the output device.

The pencil follower reads out coordinates of points, and no variable origin or scaling facilities are provided. The origin is normally situated at the near left-hand corner of the reading table and a fixed scale of 0.1 mm per digit is used, giving a work area, for example, of 9,999 digits \times 4,500 digits. Any corrections for origin and scale would be programmed into the computer. If the chart or image to be analyzed has timing marks or incremental lines along its length, these may be analyzed in several ways:

1. By taking individual readings at the intersection of the trace with the incremental markings.
2. By analyzing the trace fully, using the line mode of output and programming the computer to incorporate the required increments.
3. By using the incremental readout facility, if

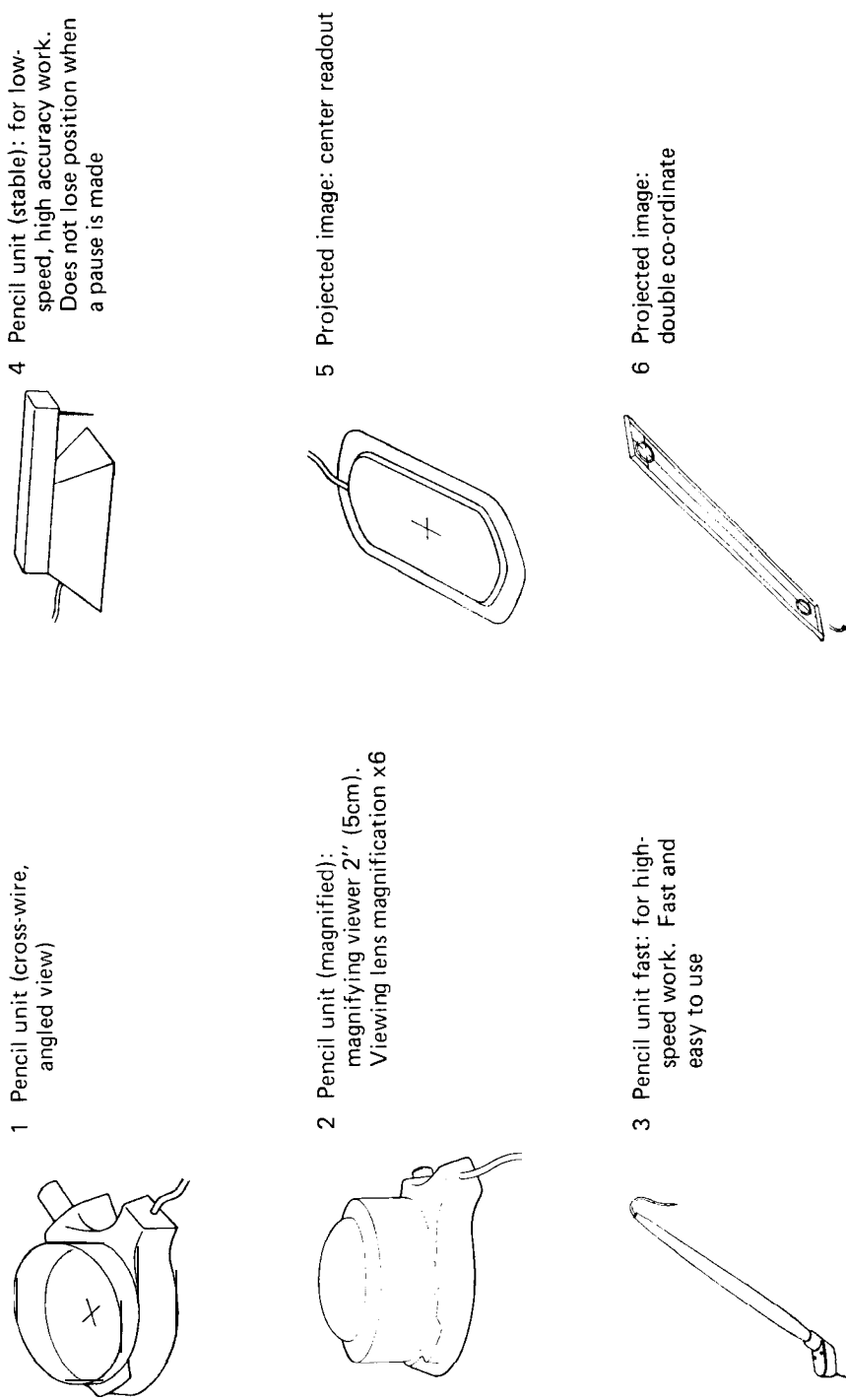


Fig. 22. The six types of reading pencil for D-MAC pencil follower (Type PF 10,000, Mark 1B).

the increments are, for example, 1 mm or multiples of it up to 1 cm.

Fig. 22 shows four different types of pencil followers used in the normal method of reading and two types used for reading of projected images.

Another device is *trace analysis equipment* that uses *two* cursors for tracing the image. The paper trace is placed in the right-hand spool holder and stretched across the length of the illuminated screen. Starting at the left-hand side of the screen, the operator sets the vertical cursor line against the first reference mark on the records and adjusts the quadrant cursor that carries the calibration curve until coincidence is obtained between the vertical line, the calibration curve, and the trace. A foot-switch is then pressed, actuating the attached output equipment—such as typewriter, punch, or plotting table (the latter using a changed scale or calibration corrections to cross-plot one parameter against another)—and the recording is thus completed. The operator moves the cursor to the next reference mark, and the whole operation is repeated.

Trace analysis equipment provides a semiautomatic system for the reduction of analog data recorded on paper or film. Average reading speeds are 1,000 to 2,000 positions per hour. Such trace analysis equipment is used as a computer input device for information that is basically in analog form. The information is converted into digital form by a converter incorporated in the device. This device may also be used if the data should remain in analog form, but the primary information has to be replotted in a different manner; an analog plotter can be attached to the trace analysis equipment.

Class 2. Devices designed to help the user with the manual description and/or registration of the image data, to be used in a direct interactive operation mode. Interactive graphic devices used in connection with video display units, like a lightpen, trackball, etc., are described in the later section entitled "Graphical Visual Displays."

As an example of a device working with a hard copy image, the microtrace information processing system will be described. Microtrace enables accurate measurements to be made by moving an electronic "free pencil" around an area or along a line. Measurements are instantly processed by a computer and printed out on an electric typewriter.

The system is simple to use. The operator who is moving the free pencil first records a few control points from the drawings, and is instructed in this procedure by a print-out from the teleprinter, the computer interrogating him with questions designed to discover information about the nature of the task

about to be undertaken. The computer is preprogrammed to make its use as simple as possible in offices where there is little existing computer expertise.

Basically, the reading pencil consists of a coil that is coupled inductively to a servodetector system situated beneath the table reading surface. Several types of pencils may be available for trace or for shape analysis and projected image work.

Class 3. Automatic image scanners as computer on-line or off-line input devices. There exist special devices designed for specific uses and which have capabilities of automatically scanning charts, often connected with some analytical evaluation method. For example, one such system is an *electronic chart reader*, which scans curves recorded previously by industrial instrument recorders on continuous paper forms and converts them into digital form. This output is then passed to the evaluation unit of the system for analysis. Because the evaluation of the chart in this unit takes place in steps corresponding to previous 15-min registrations (either selected or all of them, on the scanned chart), the system not only permits data registration and conversion, but also a very considerable reduction of the input data. An example of such a recorded chart used for evaluation is shown in Fig. 23.

There is a wide field of application for systems that need some type of automatic pattern recognition device to be used in arriving at image data needed for subsequent processing. An example of such a device is the *pattern recognition system*, which consists essentially of a set of hard-wired digital processor modules that can be combined in many ways to recognize and classify distinct objects of an image, regardless of their orientation.

The choice of source image input devices include *optical-scanning electron microscopes*, *movie and slide projectors*, *X-ray systems* and *electron probes*. The image or object is scanned electronically at the rate of one million 720-line frames a second and is converted into 650,000 picture points in a single scan. The digital equivalent of the gray value of the point is processed by the device to determine shapes, sizes, and optical densities, and to classify the objects in the image. This data can be further processed by an on-line, desk-top, or other computer, or it can be output onto computer-compatible paper tape.

An "intelligent" *computer-controlled robot system* has been designed in Japan which can read a technical drawing and, following the information read, can assemble simple machine parts. It is in fact a process-control system consisting of several units,

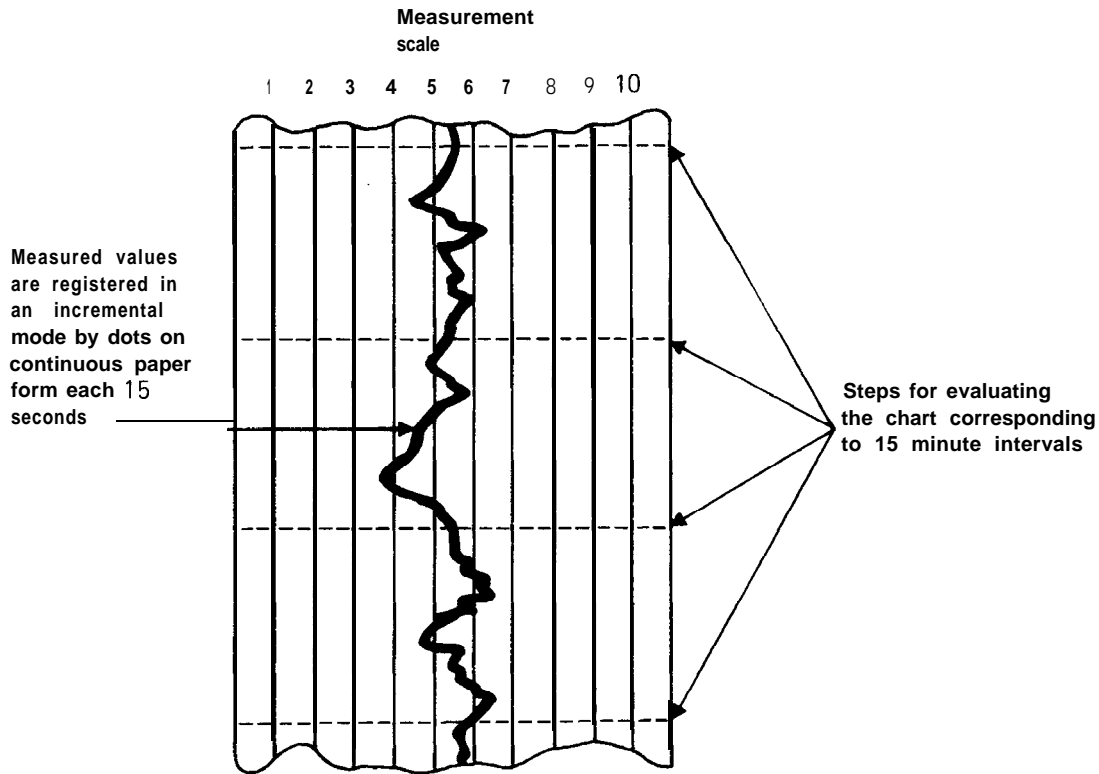


Fig. 23. Example of a chart recorded by an industrial instrument to be converted into digital form.

the "eyes" acting as optical pattern recognition devices procuring input data for the "brain," which is represented by a medium-scale computer. The "eyes" are equipped with two television cameras. The first camera surveys and selects parts to be assembled, and the second one reads the drawing (prepared in a conventional way, with special boundary lines drawn between the several different views). Computer output controls the "hands" of the robot system, which are represented by a lever-operated jaw manipulator.

The system operates in such a way that the robot first reads the drawing and then the computer calculates from this data the three-dimensional shapes of the assembly parts and their positioning in the assembly in relation to all other parts. Then the second camera looks for the corresponding part among those prepared on the feeding table by comparing their shapes with that of the computer image; when a match is found, the "hands" catch the selected part and put it to the assigned assembling place, and move it to the proper position.

Input devices used in microfilm image systems are described in the last section of this article under the heading "CIM—Computer Input from Microfilm."

ELECTROMECHANICAL PLOTTERS. Automatic plotting devices are used in conjunction with digital computers where graphic or pictorial presentation of computer data on a hard copy are meaningful and easier to use than extensive alphabetic or numeric listings. (A few examples of such simple image listings were discussed earlier under "Line Printers.") They are indispensable when the volume of graphic presentations of output data makes it uneconomical or impossible to perform the task manually.

Electromechanical plotters are impact-plotting devices; hence, they are sometimes called "ink-on-paper" or "pen-on-paper" plotters. They operate generally on the basic digital incremental principle. Decoded input commands from the computer are used to produce increments of movement in either direction along either axis, or at some angle relative to the axes. In the electromechanical ink-on-paper

plotters, the plot is produced by movement of a pen relative to the surface of the recording paper.

Electromechanical plotters generally operate in completely digital fashion and hence are drift-free. Accuracy is not dependent upon voltage stability as it is in systems that employ digital-to-analog conversion for positioning of a servomechanism. Since operation is fully incremental, there is no restriction on format. The user has complete freedom of choice in size, type, and orientation of letters, symbols, lines, and axes. Plotters are used for either on-line or off-line operation, often with any standard computer. Some of them may be used as terminals.

The product lines of major digital plotter manufacturers include several model series of electro-mechanical ink-on-paper digital plotters. Usually, they are all capable of operating in the incremental mode and some of them have added the so-called Zip Mode[®] capability. In this mode, each input plot command represents a velocity increment, and causes an increase or decrease in speed relative to either or both axes. The Zip Mode allows for high-speed plotting of straight lines and, at the same time, reduces the amount of magnetic tape required for off-line operation.

Two different command structures for the incremental mode are available with electromechanical plotters: an g-vector structure and sometimes the more precise 16-vector structure. Several increment size options are usually offered with each model, ranging mostly from 0.002 to 0.010 in.; increment size is determined by gears in the plotter. The vector diagrams in Fig. 24 illustrate the direction and incremental values possible with each of the incremental input command formats.

Plotters are of the drum or flat-bed type. The *drum-type plotters* are available in several sizes, usually a 12-in. drum and a 30-in. drum. The plot is

produced by rotary motion of the drum (X-axis) and lateral motion of the pen carriage (Y-axis). Either ballpoint or liquid-ink pens may be used. The drum-type plotter uses special chart paper rolls and can produce continuous plots up to, for example, 120 ft in length. A wide selection of paper is available. An overall view of the equipment is shown in Fig. 25 and its functional diagram is in Fig. 26.

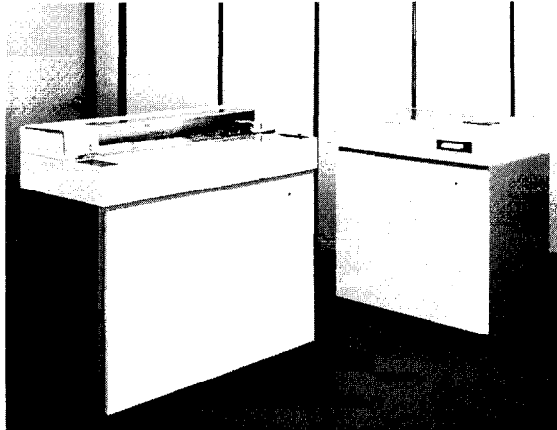


Fig. 25. CalComp 1036 drum plotter.

Flat-bed plotters are also available in several sizes, such as 31 by 34 in., 54 by 72 in., etc. (plot area). The plot is produced by lateral motion of the beam and vertical motion of the pen carriage. Either ballpoint or liquid-ink pens may be used. The flat-bed plotter provides a continuous display during plotting. It does not require special paper, and can handle a large variety of preprinted forms and special materials. A picture of a flat-bed plotter is shown in Fig. 27.

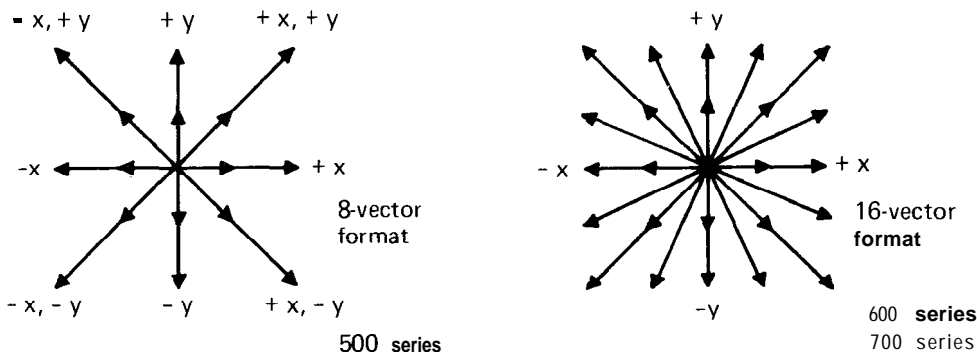


Fig. 24. Vector diagrams for CalComp plotters.

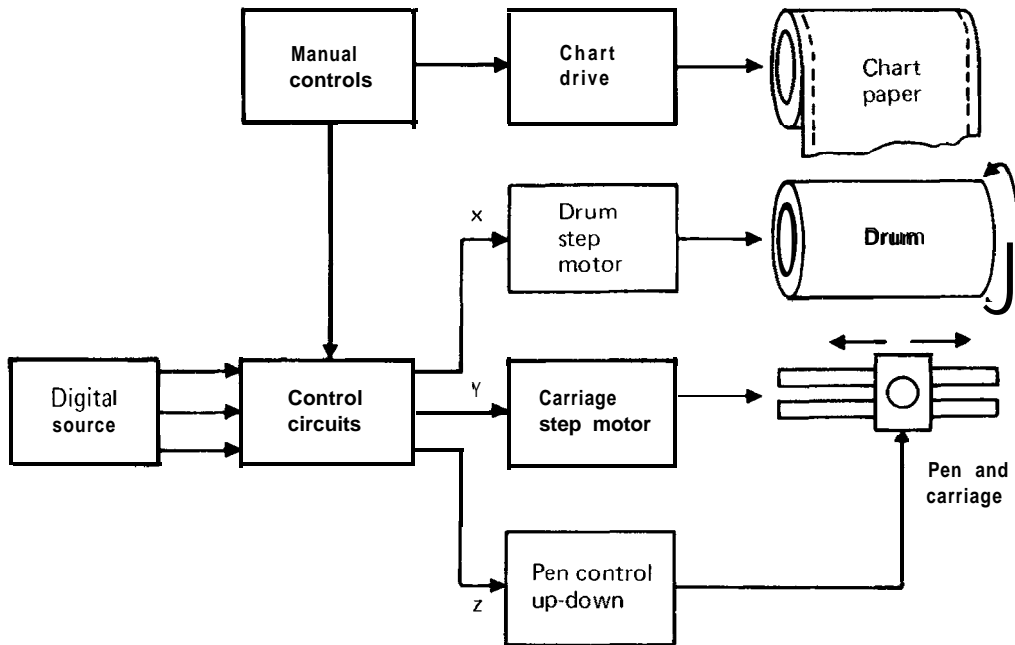


Fig. 26. Functional diagram of the CalComp 565 drum plotter.

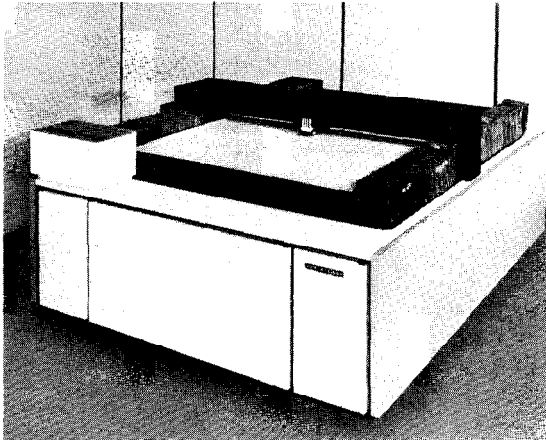


Fig. 27. CalComp 745 flatbed plotter.

GRAPHICAL VISUAL DISPLAYS. Graphical visual displays are interactive graphics devices, often called “graphoscopes.” They are entry/display devices that enable the operator/user to manipulate graphic material in a visible two-way, real-time communication with the computer.

Data Entry Means for Graphoscopes. For communication with the graphoscope, the operator has at his disposal (depending upon the specific hardware used) a lightpen, a keyboard, or other data

entry means. A *lightpen* is a photosensitive device that generally consists of a small photocell on the end of a rod (Fig. 28), but for better performance (although at higher cost) can use as moderm a highly sensitive photomultiplier tube, located inside the cabinet, to which the light is conveyed along a flexible fiber-optic light guide.

The *lightpen* has two functions: One is to say “there,” and the other is to say “do that.” Pointing a *lightpen* at the display indicates that the user wishes to say something about the part of the picture to which he is pointing. When used for drawing, the *lightpen* is pointed at the center of a small cross, called a “tracking cross,” which is displayed on the screen by a tracking program. The movement of the pen is interpreted by the computer program to actuate movement of **the** tracking cross, which follows the pen across the screen.

In addition to the lightpen, a *keyboard* similar to that of an alphascope is used for typing messages to the computer.

Switch indicators provide another means of communicating with the computer. By making use of software techniques, the operator may control the computer operation, call up frequently used data, and cause special effects to occur.

The *trackball* (see Fig. 29) performs a function similar to a lightpen. It is a phonolic ball inset in the

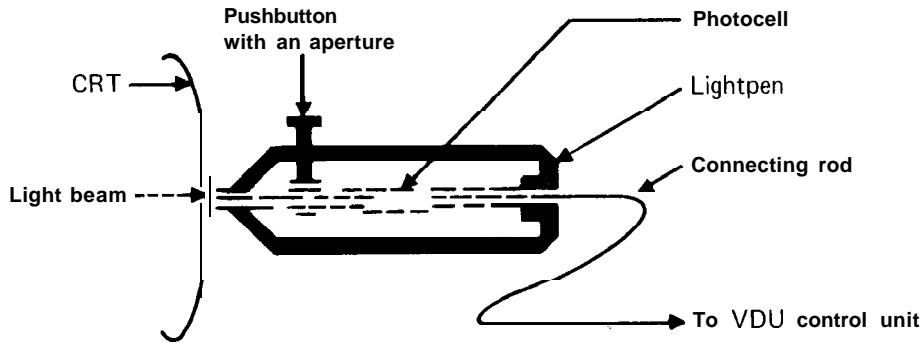


Fig. 28. Schematic diagram of a lightpen.

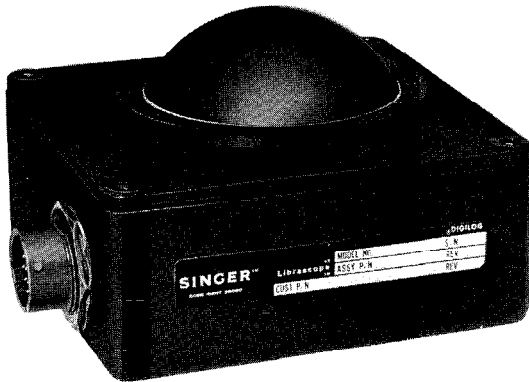


Fig. 29. Solid-state magnetic trackball.

display console. When it is invoked by the operator, a cursor appears on the display screen and follows the rotation of the trackball, continuously keeping track of the **X-Y** position. When desired, the operator simply pushes an "interrupt" button to transmit the coordinate data to the computer.

The function of a **lightpen** is sometimes replaced by a **joystick** or by a **data tablet display**. Compared with the lightpen, these devices have only limited capabilities.

I/O Functions of Graphoscopes. Unlike the alphascope, whose prime objective is fast retrieval of the desired information from the computer store, and display it onto a screen, the graphoscope is mainly used for drawing a picture, amending it, and then storing it for further processing or plotting it on a hard copy, as required. However, a graphoscope can also be used for retrieving and displaying stored pictures. Besides its graphical capability, it has to have an alphanumeric feature as well, since the graphs to be displayed have to be presented with adequate headings and notes.

A graphoscope consists of a display unit and some entry device, as discussed previously. Because of its sophisticated nature, its performance has to be monitored by a **controller** interface between the graphoscope and the computer. The controller is equipped with a vector generator, a character and symbol generator, and control logic. Sometimes arc or curve generators are also included, and an addressable buffer store is often added to speed up the overall operation of the system.



Fig. 30. User drawing a picture with a lightpen on the IBM 2250 display unit.

INPUT-OUTPUT DEVICES

The *display* unit of the graphoscope has an appearance very much like that of an alphascope (Fig. 30). The viewing screen is generally oblong, like a television screen, but sometimes is round, its diameter usually being between 14 and 21 in. Diameters as small as 8.5 in. or as large as 25 in. are known. The whole surface can be used for displaying characters, symbols, vectors, and points, generally using a raster with 1024 addressable positions in the X- as well as in the Y-axis. Some terminals use a less dense raster (e.g., 512 by 512 lines) for the same size area. Some large systems use a still more condensed raster (e.g., the CDC 274 Digigraphic display console, which has a raster of 4096 by 4096 positions over an area of more than 300 sq. in.).

The character set used generally includes approximately 64 different characters and symbols. Several VDUs have, as an option, an extension of the set to 128 characters-to meet requirements for special symbols or different character styles, as well as for displaying characters in more than one size (up to four sizes are often offered).

The number of lines needed to display the alphanumeric text is usually between 16 and 64, depending upon the size of the screen and the size of character set used. Normally, from 32 to 128 characters are displayed on a line.

In underdeveloped countries, special symbols are often used instead of the alphabet for better understanding by illiterate operators.

All functions of the graphoscope have to be done quickly and automatically. The operator should not be bothered with the problem of translating the graphic image and all the manipulating commands into mathematical terms and then into computer language. This is the task of the control unit and the computer, hardware as well as software.

In comparison with the plotter control discussed previously, the software controlling a graphoscope not only must be able to handle a static picture, but also must be very flexible to permit manipulation of the picture (lines and curves in two- or three-dimensional images). Hence, the software considerations are even more important than those of the hardware.

One of many sophisticated techniques developed in recent years is at the Brown University Computing Center in Providence, R.I., where a special program enables the computer at the center to produce a pair of three-dimensional images-differing slightly in perspective-side by side on a graphoscope. When a user looks through a special viewer (stereoscopic), he sees the two images merged into one, with the added dimension of depth (Fig.

31). The objective of this technique is to avoid building actual models, which are difficult and expensive to construct, especially in such cases as a complex refinery pipeline model in the petroleum industry.



Fig. 31. Brown University student using stereoscope to get a three-dimensional view of a Möbius strip.

If desired, *devices using microfilm cameras* are used to transfer any displayed picture onto microfilm for a hard copy representation (in one or more copies).

Most present-day interactive computer graphics systems are much more expensive than alphascope; their average cost is about \$100,000. Such a system configuration would include a small general-purpose computer (very likely a microcomputer), a display controller, a display screen, a lightpen, and a keyboard. The system may either work independently or, when a larger computer and/or data-storing facility is required, be connected to an appropriate larger system. Users of these systems are mainly large broad-based companies; they claim, however, that although it is impossible to put a profit or cost-saved figure on graphics, they could not afford to be without computer graphics. Computer-aided design means better design in shorter time, but it is not necessarily cheaper design; computer-aided management seldom means cheaper management.

The economics may change, since recently new types of low-cost graphics terminals have been introduced, but even the most reasonably priced system costs about \$8,000 for one unit, which can be used as a remote terminal via telephone lines. Such terminals use either core-refreshed CRTs or storage-

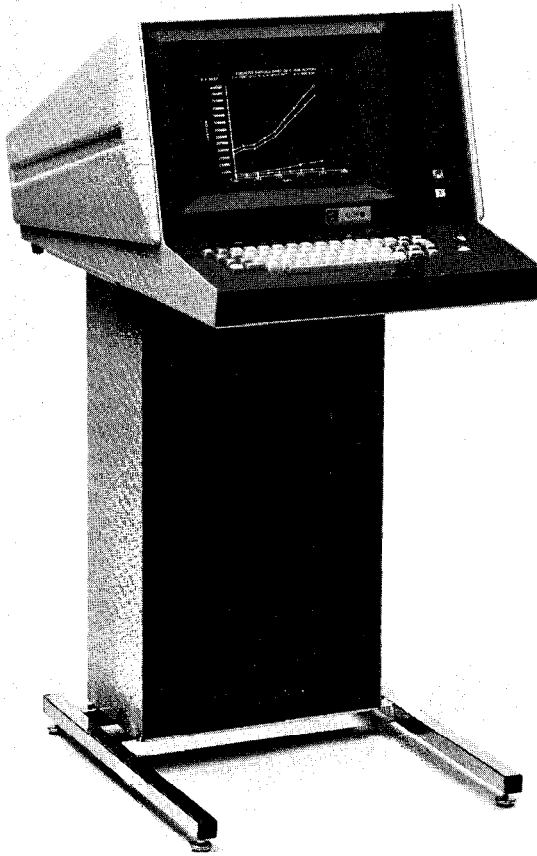


Fig. 32. Tektronix 4012 computer terminal with storage tube display.

tube displays (Fig. 32). This type of terminal looks very promising for bridging the gap between alpha-scopes and interactive graphic systems.

A half-step toward full graphical presentation has been made with the "rear-port" CRT. Permanent information, such as a map or a plant diagram, is back-projected onto the screen of the tube, leaving only the transient alphanumeric data to be handled in the conventional way.

Television Display Devices. Television-type displays are sometimes used as a less expensive solution of the image presentation requirement. Color television techniques are also applicable to such displays. A simple two-color presentation needs two separate electron guns to produce different beam energies in the CRT. A special phosphor on the CRT screen responds with red or green light, depending on the energy of the incident electrons.

There are also systems that use color displays as their main components. They may be standard

commercial 22-in. TV sets with some minor modifications. In these, the three color points give red, green, or blue color, and other hues can be obtained by mixing these colors. Four different colors can be program-controlled; red, blue, green, and white. The cursor and the *trackball* markers have orange color. In alphanumerical mode, the capacity of the screen is 25 lines, with 64 characters each. The individual ASCII characters are built upon a 5 by 7 matrix. In the graphic mode the horizontal and vertical resolution is 384 and 256 points, respectively.

A look into the future projects the use of microcomputers in driving a TV display that could be applied in personal service systems, such as home viewing, where public TV channels are becoming fully utilized*. Consider how this range of transmission could be expanded by the application of a chip microcomputer to processing a television picture. According to design standards, each TV channel has a 4.5 MHz bandwidth, with a 540-line raster painted 30 times per second. Assuming 512 lines by 512 elements per line, multiplied by 32 frames per second times four possible signals (off, red, blue, yellow-for halftone reproduction), we come up with 2^{25} bits per second. This is close to 32 million bits per second so that a microcomputer with a speed of 10 million instructions per second and 32 bits per word could drive such a display.

Since each frame is $2^9 \times 2^9 \times 2^2$, or 2^{20} bits, 2^{16} words (64K) in a computer internal store (with 5 bits a word) could hold two frames, or one frame plus instructions. Much more important would be the fact that with a computerized display, we would not need to transmit and retransmit redundant information; that is, those parts of the picture that remain the same from frame to frame could be called from computer storage.

Taking account of the fact that large areas of a picture are of solid colors and that motion from frame to frame is fairly slow and continuous, we can estimate that perhaps only 1 percent of the information would change between successive frames. This means that we could reduce the picture bandwidth by a factor of 100 and that all TV channels presently allotted could be squeezed into the space now reserved for Channel 2. This would have several major effects. The quality of the picture would be higher than the current standard. The range of a transmitter at the reasonably low frequency of Channel 2 would be much greater than at high frequencies and would practically eliminate the present poor reception in fringe areas. Given computer decoding, the cost of receivers would be substantially lower than at present. Most important,

INPUT-OUTPUT DEVICES

this application of computers to TV transmission would free for other uses nearly 400 MHz of the broadcast spectrum, a resource that is in very short supply.

MICROFILM I/O DEVICES. In the past decade, use of microfilm has expanded into a vast range of application areas connected with space-saving archival storage. This has been due not only to the general problems and pressures of information storage and retrieval, but also to the development of a more flexible range of microfilm equipment in parallel with the application of computers.

Some attempts were made earlier to coordinate the two technologies, but without decisive success. Only few years ago, with the introduction of new interfacing equipment, the situation changed rather quickly. Computer people began to look at microfilm as a technique worth considering for inclusion into information systems that need a large data base but not up-to-the-minute response time, with very limited or no need for updating, and with the possibility of storing alphanumeric information as well as graphics in a very condensed manner and for low cost. From all this investigation, new terms emerged: **COM** (Computer Output on Microfilm), and the not so much used **CIM** (Computer Input from Microfilm). Before discussing **COM** and **CIM**, the different forms of microfilm used in connection with computers will be briefly examined.

Forms of microfilms can be divided into five groups :

1. *Roll microfilm.* This is similar to 16 mm or 35 mm film, but is not perforated. The 16 mm film is used in two different ways, either with one or two tracks of images along the film. With one track, the reduction ratio applied is about 24: 1; with two tracks, the ratio is about 43: 1. The approximate doubling of the reduction ratio in fact quarters the quantity of film used. Rolls of film are loaded into cassettes, or cartridges, and used in roll-film viewers. There are already a number of double cassettes that overcome the problems of the old single cassette; rewinding is not necessary and the mechanics of the viewer are simplified.

2. *Jackets.* Short strips of microfilm (or individual frames) are inserted into clear plastic covers (acetate sleeves). Related information can be held in one jacket to provide a quick manual retrieval system, since a written reference to the identification codes it contains is put along the top of each jacket like the heading on a index card. Jackets may be updated by inserting new frames or pages of information in the plastic covers. As an example, a Bell

and Howell jacket microfilm system uses jackets that measure 5 by 8 in. and hold sixty 16 mm film frames each.

3. *Microfiche.* This is a sheet of 105 mm film carrying an orderly arrangement of micro images. The rectangular transparency of a microfiche is approximately 4 by 6 in. (about a postcard size) and can hold approximately 60 to 70 images, but sometimes holds as many as 224 full-size pages of computer output, (There is at least one printer on the market that automatically arranges the images and annotates the card with headings, allowing ease of handling and retrieval.) NCR, in its **PCHI (Photo-CHromatic micro-Image system)** uses a two-step reduction technique (the first step being a 35 mm image) that results in a reduction rate up to 250: 1. Each microfiche sheet contains more than 3,000 11 in. by 8.5 in. pages, and is generally called "ultra-fiche."

4. *Aperture cards.* These were introduced in 1945 when a microimage was inserted in a conventional 80-column punched card, which could then be sorted automatically. Normally, 35 mm film is used. Today, as many as eight images can be inserted into a single card.

5. *Microcards.* Development of new techniques also brought changes in the original idea to process (mainly sort and select) aperture cards by a machine. First, they enabled replacement of the punched code representation of the image identification data in the card by binary coded data printed with higher density, which can be read either optically or magnetically by machine. Thus, more information required for manipulation can be stored on a relatively small area of the card and processed by machine. Second, since the card no longer needs to be processed by the standard punched card equipment, its size can be enlarged. This, together with gains from the data representation change, has made it possible to enlarge the image area. The layout of these microcards (such as Filmorex, Minicard, and Magnavue) is very similar to those of the advanced type of microfiche (and may also contain image identification data),

In all these forms, black-and-white (**B** and **W**) microfilm is used, although the use of color microfilm is under study. While **B** and **W** can store many levels of information at many places by varying signal densities, from opaque through the gray scale to transparency, color microfilm could store many levels of the same print by varying not only the density but also the color.

Most of the computer microfilm-generating devices have the disadvantage of making positive microfilm, which is commonly known to be not so easily read as negative microfilm on a screen. However, a reversal film especially made for computer print-out has also been produced, at least by one manufacturer.

The arrangement of the pages on the film can usually be varied, depending on the particular unit used, but normally the choice is either the type of arrangement similar to that used in printing comic strips or to that used in movie film, with reduction ratios of around 24:1 and/or 43: 1. The choice depends on the sophistication of the equipment.

COM. The name "COM" should be related to the overall concept of accessing computer-based information via microfilm. However, more often the term is used to refer to the hardware device that generates the microfilm. Depending upon the technique used, a COM system usually consists of three sections (all of which may be accommodated in one cabinet, or kept as separate units):

1. A *tape drive* in an off-line installation, or a *computer* if the mode of operation is on-line, provide data to be microfilmed.
2. A *control unit*, including a buffer store for speeding up the throughput, a symbol generator for printing alphanumerical and special symbols, a vector generator for drawing graphs, and control logic. The latter coordinates and directs the action of all system elements to achieve the end result of exposed microfilm. It selects the input tape, what is to be recorded, and in what size and position of the "page" it should be located, and when the film is to be advanced to the next frame. It also controls the coding (if any), tape-error conditions, reread; and frame marking to show unreadable characters.
3. The *microfilm recorder* with the microfilm transport and positioning section, and the optical system used for the recording and developing the frame.

In some systems, if hard copy is required, recordings on microfilm can be suppressed and instead the image can be copied in the desired form.

Many COM plotters and/or printers have the ability to superimpose on the printed image one of several program-selectable forms, giving a combined image on microfilm or on hard copy stationery.

The microfilm produced by the COM system can also be used in a conventional way. If hard copy is required, high-speed copying devices using microfilm frames as a copying matrix for producing one or

more printed copies are available. Similarly, another device is applied when one or more duplicates of the microfilm are required. The kind of device to be used is determined also by the requirement to ease the selection operation in the information retrieval process. At least two manufacturers produce equipment for automatic microfilm-stored information retrieval.

Generally, all COM systems on the market are offered for off-line operation, but many of them can work on line with the computer. However, there is at least one system that operates in conjunction with the computer in on-line mode only.

Nearly all COM devices translate data from magnetic tape to microfilm via a CRT presented to a microfilm camera, although there are variations of this scheme, such as an electron beam recording directly onto the film or a fiber optics system to present the character image to the camera on line from the computer.

COM hardware on the present-day market is divided into three categories: COM graph plotters, COM printers, and COM plotter/printers: these devices are sometimes known by other names.

COM graph plotters use a design technique very similar to that of the pen-on-paper plotters. Drawings are composed by the creation of incremental moves in the X- and Y-axes by deflection of an electron beam on a CRT. A third move required, equivalent to the raising and lowering of the pen, is achieved by blanking and unblanking the beam. The drawing areas are typically on a raster of 3,000 to 4,000 positions with a standard 15X magnification of the 35 mm microfilm frame, which would give the equivalent of a 0.005 in. increment size resolution on a finished drawing of 11 by 17 in. The plotting speed varies with different models and can be up to 500,000 increments a second. Fig. 33 shows a representative device of this group.

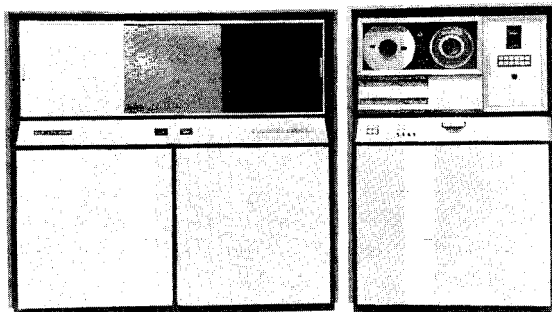


Fig. 33. CalComp 1670 microfilm plotting system.

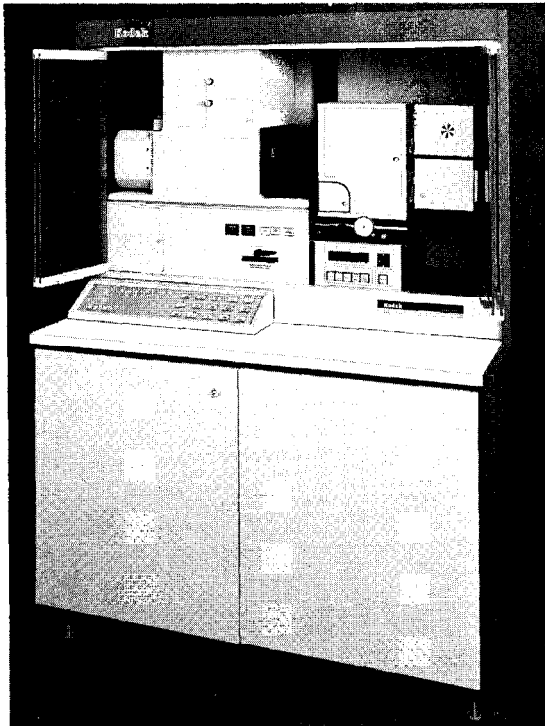


Fig. 34. Kodak KOM-90 microfilmer.

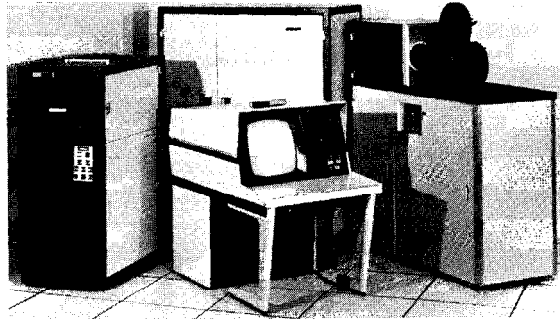


Fig. 35. CDC 280 recorder and display system.

These plotters can plot graphs as well as alpha-numerical characters. They are, however, not suitable for use as high-speed commercial printers.

COM printers (Fig. 34) use a symbol generator or other technique to generate characters to be recorded on microfilm and then printed out (all of them, or selectively). Since the microfilm image of the information stored has a form similar to that of a page printed by a high-speed line printer, the *COM* printers are often called "high-speed page printers." The printing speed varies with the different models

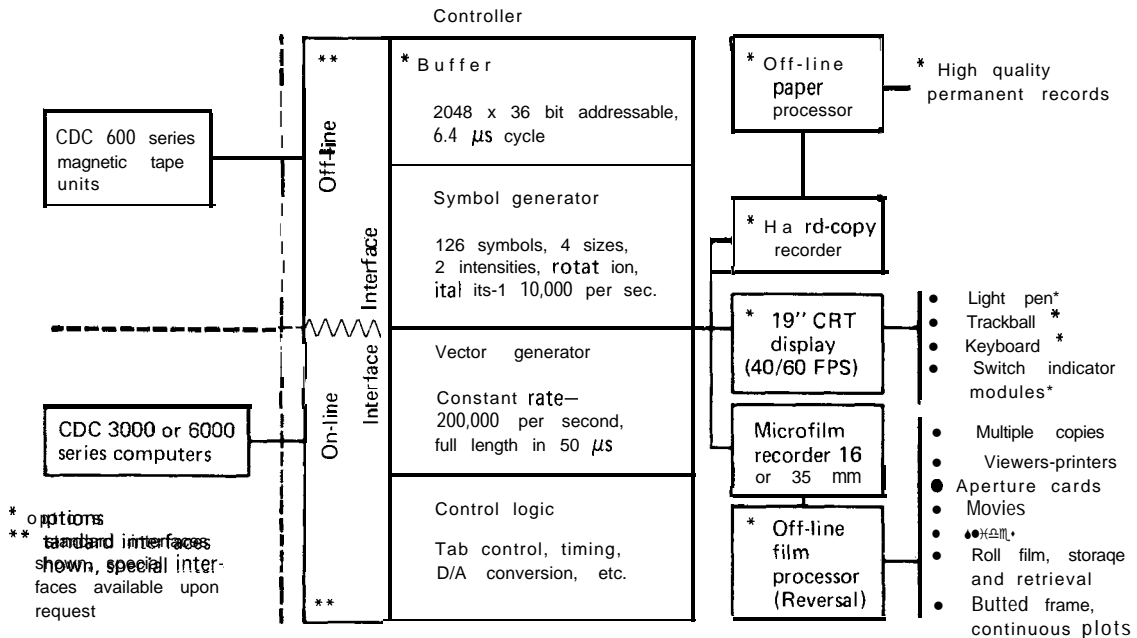


Fig. 36. Schematic of Control Data 280 recorder and display system.

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	:	=	≠	≤	≥	[
2	+	A	B	C	D	E	F	G
3	H	I	<	.)	≥	~	;
4	-	J	K	L	M	N	O	P
5	Q	R	v	\$.	↑	COMING	>
6	BLACK	/	S	T	U	V	W	X
7	Y	Z]	,	(→	≡	Λ

UPPER CASE

	0	1	2	3	4	5	6	7
0	∇	1	2	3	ψ	ρ	τ	ϕ
1	ω	a	β	6	Σ	σ	u	π
2	θ	a	b	c	d	e	f	g
3	h	i	z	3	'	'	•	?
4	±	j	k	l	m	n	o	p
5	q	r	/	a	'	Δ	COMING	f
6	BLACK	+	s	t	u	v	w	x
7	y	z	~	8	0	Δ	ϕ	v

LOWER CASE

Fig. 37. CDC recorder and display system; machine-generated symbols and characters.

used and can be as high as 40,000 lines per minute; there are at least three manufacturers who offer equipment attaining or coming close to this speed. COM printers do not print graphs.

COM plotter/printers have features of both the plotters and printers mentioned above. Hence, they can be used in both ways. A representative of this category is shown in Fig. 35. The accompanying diagrams show the possibilities of such systems in a hardware configuration (Fig. 36), in a machine-generated symbol and character set (Fig. 37), and in control word formats (Fig. 38). These plotter/printers which may also use VDU, lightpen, etc., can be considered representative of true *interactive microfilm processing*. Other interactive systems use the computer only for calling up the next required image to be projected onto the screen for viewing by the user, but not for changing the plotted image.

To ease later manipulation with the microfilm, mainly in the selective retrieval operation, the COM system generally prepares, during the COM process, the identification codes for frames, reels, etc., as necessary.

CIM. The CIM system may be regarded as the inverse of COM. However, in comparison with COM, CIM is still far more in the research and development stage, with just a handful of systems already installed. The decisive factor that causes difficulties in the design of a CIM device concerns the level of universality of the images to be read, as

well as the patterns to be recognized. CIM devices usually can work on line as well as off line. From the few devices available, we will describe three to give the reader some idea of the current range of capability.

The IBM 2281 film scanner may be used on line with IBM 360 and 370 systems. Negative images (clear lines on dark background) contained on 35 mm unsprocketed roll film are converted to input data by moving the CRT beam of the IBM 2281 film scanner over the surface of the film in the scanning pattern specified by the program. The complexity of input information that can be handled depends upon the power of the programs written to interpret the data generated by the 2281. Examples of acceptable images are drawings, charts, maps, and graphs.

A frame size up to 1.2 in. square may be scanned before it is necessary to move the film. The film is advanced or backspaced (incrementally, or a full frame at a time) under program control, or manually. Registration of the film image can also be done manually or under program control.

The Information International Incorporated **GRAFIX 1 CIM** configuration is in itself a large computer system containing two processors, the binary image processor (BIP) and a PDP-10 computer with a two billion bit disk store, six tape drives, a number of operator consoles, and a film scanner optical system. The film to be read is positioned frame by frame between a programmable CRT (light source) and a photomultiplier (the image detector).

INPUT-OUTPUT DEVICES

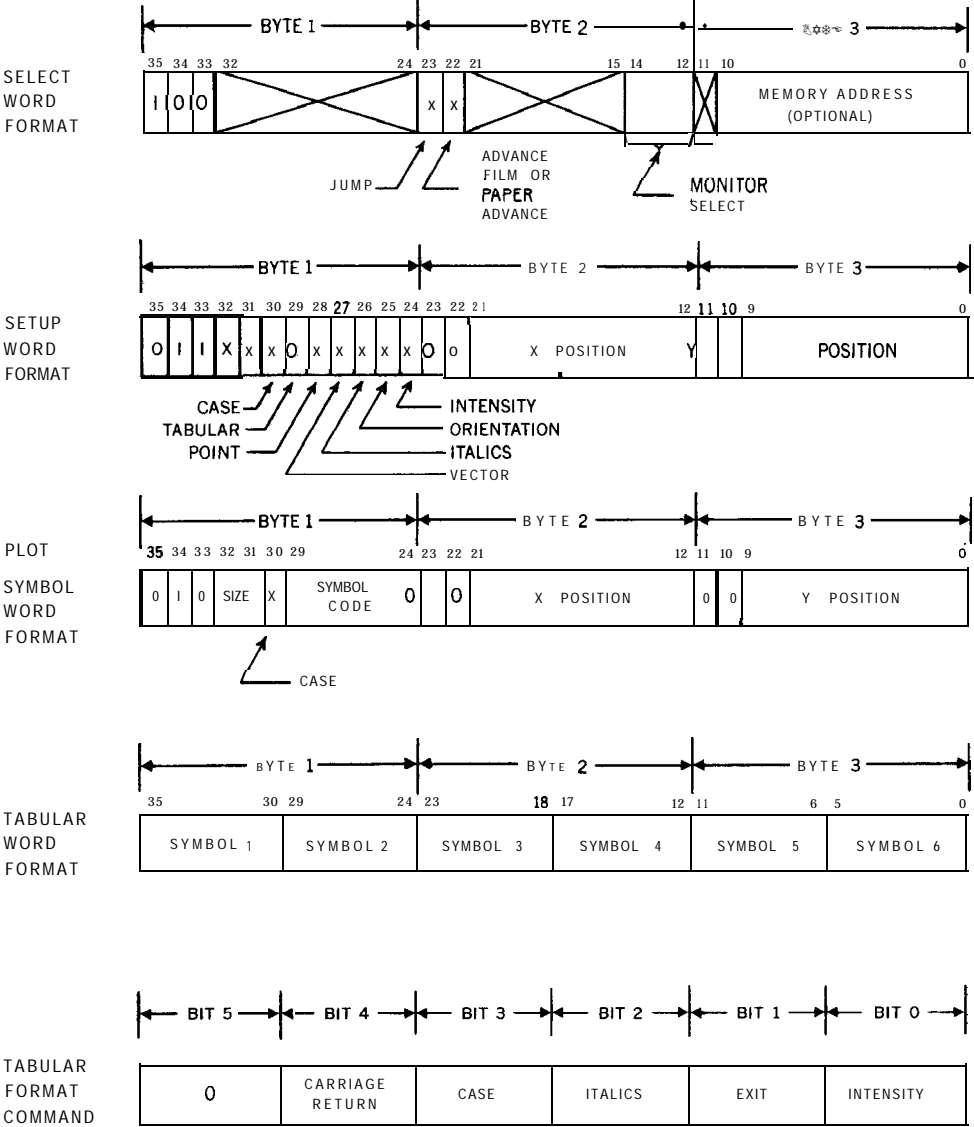


Fig. 38. CDC recorder and display system: control word formats.

Under control of the BIP, over a billion points on each frame of the film are examined as potential image constituents. Each character on the film will generate a unique dot pattern within the matrix enclosing it. This pattern is compared with the stored patterns until a match is achieved. Any unrecognizable characters (typically 1: 10,000) are displayed on one of the operator consoles for verification. If a previously unknown character represents a member of a new font, the GRAFIX 1 system "remembers" the new style for future reference.

Each font stored in the GRAFIX 1 may consist of up to 92 symbols, and to avoid an average of 46 comparisons in the BIP prior to character identification, the PDP-10 attempts to predict forward characters, using known statistical distributions. The manufacturers claim that this reduces the average number of comparisons in the BIP to 4.6 before a match is achieved. Reading from a known font, the GRAFIX 1 is supposed to input 2,000 chps from microfilm to its backing store.

The GRAFIX 1 is basically a pattern-recog-

nition device and, because of its high degree of resolution, has been used for some interesting experiments, including removing scratches from old Donald Duck cartoons and identifying missile sites from satellite photographs.

The *Oxford Precision Encoding and Pattern Recognition device* (PEPR) was developed for the purpose of identifying and measuring tracks left by the reaction of nuclear particles in hydrogen bubble chambers. Over two million frames of bubble chamber events photographed on 50 mm film have been processed automatically in the first two years of PEPR operation. More recently, PEPR has been used as a converter to read basic rainfall charts recorded by autographic rainfall recording instruments (produced in hundreds of locations throughout the United Kingdom for periods up to 50 years) and to convert them to a complete digital record of rainfall in the U.K. on magnetic tapes for later processing on a computer.

J. NECAS

INPUT-OUTPUT INSTRUCTIONS

For articles on related subjects see **CENTRAL PROCESSING UNIT; CHANNEL; INPUT-OUTPUT CONTROL SYSTEMS; INPUT-OUTPUT DEVICES; and MACHINE INSTRUCTION SET.**

Input-output (I/O) instructions cause transfer of data between peripheral devices and main memory, and enable the central processing unit to control the peripheral devices connected to it.

Prior to any explanation, a rudimentary model of the logical structure of an I/O setup is necessary. An important point to note is that the model used here is purely logical; i.e., in any actual computer organization, some of the units to be mentioned may be physically nonexistent. Their function, in such a case, will be integrated into the other existing units. This will not change the description of the I/O procedures and operations that will be presented in this article.

The model we use is illustrated in Fig. 1. The central processor and its memory are connected to channels. The number of possible channels is variable. Each one of them has an identifying name (i.e., number). Each channel can accommodate a number of peripheral device controllers. Each controller will control one or more identical, or very similar, devices such as line printers of different speeds, disks, and drums.

In the area of I/O processing, the distinction between hardware and software is extremely vague. In certain cases, vendors of computing equipment include in their hardware manuals a description of I/O instructions, which in reality are parameters to subroutines that incorporate the actual hardware I/O instructions. The questions of the physical existence of channels and controllers must also be carefully dealt with. The reader is advised to keep these gray areas in mind when trying to apply the following discussion to an actual computer.

NOMENCLATURE. Let us now establish some nomenclature. The sequence of I/O operations needed to perform an actual data transfer will be called an "I/O procedure." In an I/O procedure, all devices present in the I/O setup (i.e., the central processor, the channels, and the controllers) take

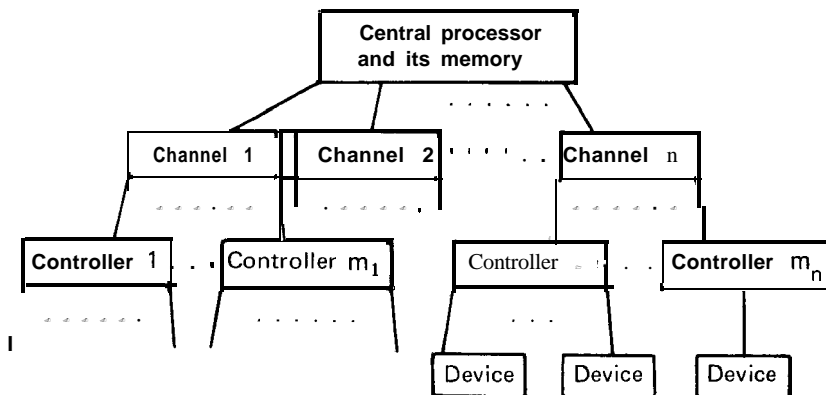


Fig. 1. Model of an I/O setup.

INPUT-OUTPUT INSTRUCTIONS

part. They operate as independent processors, each performing its own type of operation. We distinguish between I/O *instructions* performed by the central processing units, I/O *commands* performed by the channel, and I/O *orders* performed by the controllers. The degree of independency and concurrency of these operations will be dealt with later.

As usual when introducing nomenclature, it is important to note that different vendors use different words for the same concept. We will follow the more or less established nomenclature, but will introduce synonyms in the proper places.

I/O OPERATIONS. I/O operations are of two classes: control operations and data transfer operations.

Control operations perform the following tasks:

- 1. Establish the *data path* between the main memory and the peripheral device.
- 2. Check to verify that the path is legally established and that all devices in the path are operational.
- 3. Diagnose the success or failure of all data transfer and control operations.

Data transfer operations initiate and terminate the actual data transfer through the preestablished path. These classes cover the whole spectrum of I/O operations.

I/O INSTRUCTIONS. These are regular machine instructions in one of the formats acceptable to the computer. They are decoded and performed by the

central processor in the same manner as any other instruction, such as an arithmetic instruction. Examples of such instructions are: **START** I/O (available on IBM/370 and XDS Sigma, among others), which initiates a channel operation; **TEST** I/O, which returns status information about the conditions on an I/O path; **HALT** I/O, etc. The number of basic I/O instructions is usually low, but there are very many variants, which may have different meanings for different devices. These variants are usually included in the address fields of the instruction or deferred to the channel command.

While performing these instructions, the whole central processor is tied up, in the same way that any other type of instruction ties up the CPU.

CHANNEL COMMANDS. Channel commands, sometimes referred to as “channel control words,” or I/O descriptors, are bit strings that contain control information for the channel. They are interpreted by the **channel**, which can therefore be viewed as an independent processor whose instructions are the channel commands, and which is operating in parallel to the central processor. From such a description of the channel, it is clear why the role of the channel can, in certain configurations, be performed by the central processing unit.

The channel commands can be either contained in arbitrary or fixed memory locations (as in most large- and medium-scale computers), or contained in special registers (as in most minicomputers). In each case, the channel operation is initiated by a central processor instruction that passes to the channel the location of the channel commands, or which notifies

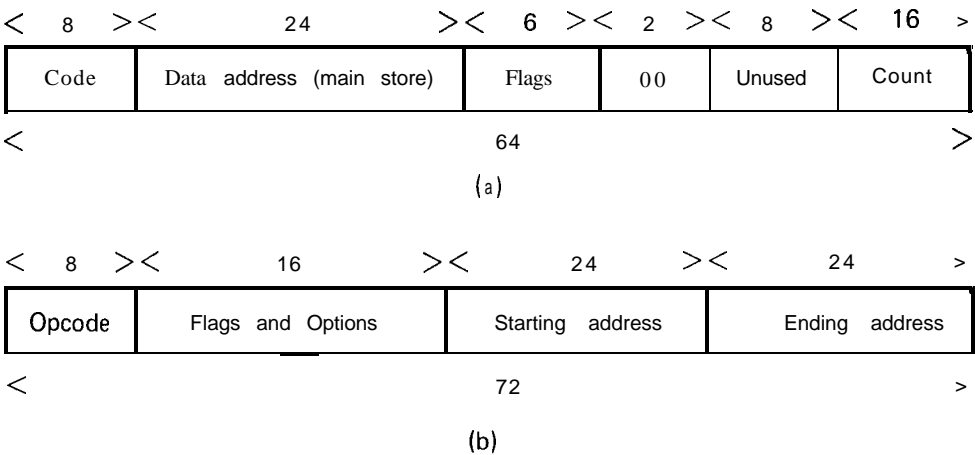


Fig. 2. Channel commands for (a) IBM/370, and (b) Burroughs B2500 (where the ending address is not always necessary). Field lengths are given in bits in both (a) and (b).

the channel to start under the assumption that the channel commands are already resident in a predefined, fixed memory location or in predefined registers.

The structure of channel commands is very similar to the structure of regular machine instructions.

In Fig. 2 are two particular examples of channel commands, taken from IBM and Burroughs. The meaning of the various fields is almost self-explanatory. The code (or "opcode") is the actual operation to be performed by the channel: **READ**, **WRITE**, **READ BACKWARD** (in the case of magnetic tape), **MOVE** (the recording) **HEADS** (in the case of disks), etc.

The flags and options are usually short fields, sometimes as short as one bit, indicating specific demands and conditions. These may include indications of the course of action to be taken on normal or abnormal completion of the channel command, additional information needed to support some opcodes, enabling or disabling options like command and data chaining (see below), modes of automatic character conversion (if applicable), etc.

The starting address and count, or the starting and ending address, serve to identify the data on which the operation is to be performed. In certain cases, where the amount of information is limited by the nature of the device (i.e., the length of the line in a line printer), it is enough to indicate the beginning of the information.

Once the channel is started, it processes its own commands, which cause the transfer of data to and from the peripheral device controllers. This data, in turn, can be interpreted by the controller, either as an actual data item or as an order to the peripheral device controller. The distinction between data and order can be done in different ways. The transferred item may have identifying information associated with it, or the sequence of arrival of the items will define them to be either orders or actual data.

I/O ORDERS. Orders to controllers may be: **START**, **STOP**, **TRANSFER Status** information, **GET** a data item, **MOVE HEADS**, **REWIND**, etc. The orders obviously reflect the nature of the controlled device.

Each step in the sequence instruction-command-order causes, in addition to its normal operation, the creation and storage of status information. This information is usually of two types. One is the setting of the condition codes or status bits of the central processor itself, as expected after each CPU instruction. The other is the creation of a channel status word, sometimes also referred to as "result descriptor." Whereas the condition codes describe the state of the CPU, the channel status words

describe the control information that is presented to the channel by the controller (together with data specifying the device itself), and which describes the status of the channel itself.

The structure of the result descriptors varies so widely among manufacturers that examples would be more misleading than beneficial. The reader is advised to consult the manual of the computer in which he is interested.

The various status information items are used to determine the success, or failure, of the I/O process. In the case of failure, the program that initiated that I/O operation can take either corrective or merely diagnostic steps.

A complete I/O procedure will therefore consist of the following steps:

1. Prepare a set of channel commands which will cause the proper set of orders on the device to be activated.
2. Issue instructions that will activate the channel. In this sequence of instructions one should first check to see if the channel is available, i.e., not busy with previous operations or is physically disabled.
3. After completion of the I/O operation, check for success and take necessary steps in case of failure.

Again, the interested reader is strongly advised to follow such a procedure in detail on a particular computer. Procedures of this type are sometimes called "I/O drivers."

We conclude with a number of disconnected remarks and notes. In an actual I/O process, it may be desirable to perform a whole program built from channel commands. The sequencing through the program can be driven by the end of each command, or by the exhaustion of the data to be transferred, without the command being actually finished. These two methods of sequencing channel commands are called "command chaining" and "data chaining," respectively. Not all computers possess this capability. When present, this option is controlled by the flag and option fields of the channel command.

In actual computer systems, the channels are sometimes physically integrated into the CPU. It also happens more and more frequently that the controllers are integrated into the devices themselves. This by no means changes the description of the I/O procedures. The channel commands merely turn into computer instructions. The actual transfer of data, made by the channel, will be done by the central processor hardware. Whether this process

INSTITUTE FOR CERTIFICATION OF COMPUTER PROFESSIONALS (ICCP)

will, or will not tie up the computer, is dependent on the sophistication of the hardware.

There exist computer configurations in which not only are the channels not integrated into the CPU, but (as in CDC CYBER 70 series) are turned into full-fledged computers. In this case, one needs a whole interpretive layer of software in these computers to interpret the I/O request posted by the central processor. In other solutions to the question of channels (used, for example, by IBM and Burroughs), the vendors supply factory microprogrammable channels which, on one hand, have the advantage of programmable computers, but on the other hand do not burden the user with software maintenance.

Finally, it is necessary to indicate that no matter which of the former hardware alternatives is present, the I/O process in basically an asynchronous one in which the channels, or their variants, are operating in parallel with the CPU. Thus, one needs a synchronization procedure by which the CPU and channel operations are coordinated. This procedure is indicated by the flag fields and involves either interrupts or polling.

G. FRIEDER

INSTITUTE FOR CERTIFICATION OF COMPUTER PROFESSIONALS (ICCP)

For articles on related subjects see ASSOCIATION FOR COMPUTING MACHINERY; DATA PROCESSING MANAGEMENT ASSOCIATION; and PERSONNEL IN COMPUTERS.

The Institute for Certification of Computer Professionals (ICCP) is an organization of computing societies, established in 1973, for the purpose of sponsoring activity in the areas of testing and certification of knowledge and competence of computing personnel. It is intended to pool the resources and interests of individual societies so that ultimately the full attention of the industry may be focused on the vital tasks of developing and recognizing qualified personnel.

The purposes of the Institute are:

1. To foster, promote, develop, and conduct scientific inquiry and research into any of the several

activities related to the development and recognition of knowledge and competence among personnel in the computer and information systems industry.

2. To foster, promote, develop, and conduct scientific inquiry and research into standards of good practice,

3. To formulate and administer testing and evaluation programs designed to determine the aptitude, level of knowledge, and competence of individuals engaging in, or desiring to engage in, disciplines directly related to applied computer and information science.

4. To foster, promote, and develop internationally the purposes of the corporation, including without limitation (i) the establishment of reciprocal standards with, and reciprocal membership for and cooperation with, organizations having similar aims and purposes; (ii) the establishment of international standards of good practice in the worldwide computer and information systems industry; and (iii) the formulation and administration of reciprocal testing and evaluation programs.

How **Established.** ICCP was incorporated as a not-for-profit corporation in the State of Delaware on Aug. 13, 1973. Its establishment was the outgrowth of several years of study by committees of the Data Processing Management Association (DPMA) and the Association for Computing Machinery (ACM) during which the concept of a "computer foundation" to foster testing and certification programs was formulated. An open invitation was extended to other societies to support an organizational period. The organizations that served on the Computer Foundation Organizing Committee and then became charter members of the Institute were:

Association for Computing Machinery
Association of Computer Programmers and Analysts
Association for Educational Data Systems
Automation I Association
Canadian Information Processing Society
Data Processing Management Association
IEEE Computer Society
Society of Certified Data Processors
Society of Professional Data Processors

The Organizing Committee was chaired by John K. Swearingen, CDP, of DPMA and Fred H. Harris of ACM, each of whom then became, respectively, the first president and vice-president of the Institute.

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

On January 1, 1976, Harris succeeded Swearingen as president of ICCP.

Organizational Structure. The Institute is governed by a Board of Directors to which each member society designates two directors. Officers of the Institute are elected from the Board at its annual meeting and include a president, vice-president, secretary, and treasurer. The officers constitute an Executive Committee, which may act for the Board between its regularly scheduled quarterly meetings.

Three principal standing committees provide advice to the Board and assist in the management of the Institute: (1) the Program Committee conducts research into the need and feasibility of new projects to be undertaken by the Institute; (2) the Public Information Committee publicizes the programs of the Institute and its concepts; (3) the Budget and Finance Committee prepares the budget for the Board's review and approval, and provides advice on the fiscal affairs of the Institute.

As programs are initiated by the Institute, councils will be established to oversee them and to provide the necessary competence to insure high standards. Councils have policy-making powers, as well as responsibility for quality control, within the appropriate domain of their programs.

Presently, the only council is the Certification Council, which has jurisdiction over the testing and certification programs acquired from DPMA and is responsible for the Certificate in Data Processing (CDP) examination described in the next section.

Programs of the Institute. In these first years of its operation, the Institute's highest priority is the improvement of existing certification programs and the establishment of new examinations for various specialties. In 1974 the Institute acquired the testing and certification programs of DPMA, including the Certificate in Data Processing (CDP) examination, which DPMA had begun in 1962 and had administered since then. Since 1962 the CDP examination has been offered annually at designated testing centers, usually colleges and universities.

All candidates for the CDP examination must have at least 60 months of full-time, or equivalent part-time, work experience in a computer-based information systems environment. Candidates may submit college level academic experience for evaluation as partial fulfillment of the current experience qualifications. The amount of credit allowed is determined by the CDP Credentials Committee.

The present CDP examination requires one day to complete, and consists of five sections: Data

Processing Equipment, Computer Programming and Software, Principles of Management, Quantitative Methods, and Systems Analysis and Design. Any qualified person may take it, and every candidate must successfully complete all five sections to receive the Certificate.

F. H. HARRIS

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS— COMPUTER SOCIETY (IEEE-CS)

For an article on a related subject see
**AMERICAN FEDERATION OF INFORMATION
PROCESSING SOCIETIES.**

Purpose. The IEEE Computer Society was formed to advance the theory and practice of computer and information processing technology. Its objectives are to promote cooperation and exchange of technical information among its members. To achieve this, the Society holds meetings for the presentation and discussion of technical papers, publishes technical journals, and through its chapters and technical committees studies and provides for the professional needs of its members. The scope of the Society encompasses all aspects of design, theory, and practice relating to digital and analog devices, computation, and information processing.

How Established. The IEEE Computer Society was so-named in 1972, having originated in October 1951 as The Computer Group of IRE (Institute of Radio Engineers), which on Jan. 1, 1963, merged with the American Institute of Electrical Engineers and became the Institute of Electrical and Electronics Engineers (IEEE). The IEEE represents some 160,000 electrical and electronics engineers throughout the world.

With so many special interests among its members, it was natural for members who wished to concentrate in one area of electronics, or who wanted to exchange knowledge with those of similar interest, to create special-interest groups. The Computer Society with some 21,000 members, is one of these special-interest groups. The IEEE headquarters is located at 345 E. 47th Street, New York, New York 10017.

INSTRUCTION

Organizational Structure. The IEEE Computer Society has a Governing Board consisting of a maximum of 23 voting members. The officers include the President, one or more Vice-Presidents, the Junior Past-President, and 20 elected members of the Board. The Board annually elects the President and Vice-Presidents from its membership. The President-elect appoints, from voting members of the Board, a Secretary and a Treasurer for a one-year term coextensive with his term. The President, under direction of the Board, has general supervision of the affairs of the Society.

Technical Program. Members of the IEEE Computer Society receive *COMPUTER* magazine, "the voice of the computer systems design profession," which contains tutorial and survey articles, practical applications ideas for the computer professional, a repository of yet unpublished papers, and various other pertinent departments. In addition, they have the choice of receiving the *Transactions on Computers*, which contains papers of archival quality on the theory, design, and practices related to digital and analog computation and information processing, or the *Transactions on Software Engineering*, which contains archival research papers on all aspects of the specification, development, management, test, maintenance, and documentation of computer software, or the *Journal of Solid States Circuits*, which covers devices and systems affecting circuit design.

The IEEE Computer Society sponsors two annual Computer Society Conferences, and cosponsors the annual AFIPS National Computer Conference,

The IEEE Computer Society's technical committees include: Computer Elements, Computer Architecture, Computer Communications, Data Acquisition and Control, Design Automation, Fault-Tolerant Computing, Operating Systems, Packaging, Pattern Recognition, Simulation, and Switching and Automata Theory. Their aim is to promote technical excellence in specific areas by sponsoring seminars, symposia, and sessions at professional conferences.

Other services of the IEEE Computer Society include microfiche and magnetic tapes containing accumulated bibliographic data, data sets to enable researchers with limited resources to obtain access to data bases of other researchers, and the Distinguished Visitors Program, which arranges for leading computer professionals to speak to local chapters of the Society and educational institutions.

Officers of the IEEE-CS since its inception (Oct. 9, 1951) include the following persons:

CHAIRMEN

Jean H. Felker, 1953-1954
H. T. Larson, 1954-1955
Jerre D. Noe, 1955-1957
Werner Buchholz, 1957-1958
Willis H. Ware, 1958-1959
R. O. Endres, 1959-1960
A. A. Cohen, 1960-1962
W. L. Anderson, 1962-1964
K. W. Uncapher, 1964-1965
R. I. Tanaka, 1965-1966
Samuel Levine, 1966-1967
L. C. Hobbs, 1968-1970
E. J. McCluskey, 1970-1971

Name changed to IEEE Computer Society on Jan. 1, 1971.

PRESIDENTS

E. J. McCluskey, 1971
A. S. Hoagland, 1972-1973
S. S. Yau, 1974-1975
D. B. Simmons, 1976-

I. L. AUERBACH

INSTRUCTION. See MACHINE INSTRUCTION SET; and PRIVILEGED INSTRUCTION.

INSTRUCTION COUNTER. See PROGRAM COUNTER.

INTEGRATED CIRCUITRY

For articles on related subjects see COMPUTER CIRCUITRY; and LOGIC DESIGN.

After Bell Telephone Laboratories announced the invention of the transistor in 1948, transistors were manufactured as discrete components in the same way as other electronic components such as diodes and resistors. Logic circuits were built from these discrete components by mounting them on a board with metal-wire connections or on a printed

circuit (PC) board. Since transistors consume less power and require less space than vacuum tubes, miniaturization of circuits (which is also referred to as the technique of "microelectronics") has been continuously pursued by circuit engineers. The advantages of microelectronics reach far beyond the apparent reduction of size and weight. They, in fact, improve the reliability of the circuits so much that a drastic increase of system complexity can easily be accommodated.

The earlier semiconductor *integrated circuits* attempted by electronic industry were in the form of the *hybrid integrated circuits* (Warner, 1965). Separated transistors and diodes, resistors and capacitors were mounted on a metallized ceramic substrate. Individual devices were interconnected by a wire-bonding technique. The next important step in the history of semiconductor integrated circuits was the development of the planar transistor. Before the planar technique was developed, junction transistors were fabricated by batch processing in a mesa structure, in which the base layer of a transistor was mounted on top of the collector layer, with the emitter region diffused into the base layer. The surface of mesa transistors is extremely sensitive to ambient conditions, and the product yield of good transistors is usually low.

In the planar approach, the base region is diffused into the collector layer and the emitter region is, in turn, diffused into the base region. The base collector junction of the planar surface is protected from possible contamination of any foreign matter by a layer of silicon dioxide. The metallization (i.e., wiring) patterns, which interconnect various electronic components, can be easily batch-processed by depositing them on top of the silicon dioxide layer. In order to make complete integrated circuits, resistors and capacitors are also fabricated on the same tiny piece of semiconductor material. These tiny pieces of semiconductor are usually referred to as "chips."

When all electronic components like diodes, transistors, resistors, and capacitors are fabricated in a single chip of semiconductor, the name "**monolithic integrated circuits**," or monolithic circuits, is used to distinguish this circuit from hybrid integrated circuits. As the state-of-the-art progressed, the complexity of monolithic integrated circuits kept increasing. The terms medium-scale integration (MSI) and large-scale integration (LSI) have been introduced. The generally accepted definitions of MSI and LSI are as follows: Medium-scale integration technology refers to the fabrication of inte-

grated circuits with circuit complexity in the 10 to 100 logic-gate range. Large-scale integration technology refers to the fabrication of integrated circuits with circuit complexity of over 100 logic-gate range. A typical logic gate that performs logic functions has approximately half a dozen components (i.e., transistors, diodes, resistors, etc.).

Although a clear-cut dividing point between MSI and LSI in logic-gate counts is provided in the preceding definitions, MSI and LSI technologies actually have no clear-cut boundaries. With some increase in chip size, MSI photolithographic tolerances can be utilized just as well for LSI chips. Both bipolar circuit configurations such as transistor-transistor-logic (TTL), and MOSFET (metal-oxide-semiconductor field-effect transistor) circuit configurations such as the p-channel FET inverter can be used in either MSI or LSI technologies.

Computer logic circuits, in integrated circuit (IC) form, have the advantages of smaller size, lower power dissipation, shorter interconnection delay, and lower cost than those in discrete device form. The IC logic circuits, however, also have some drawbacks (Warner, 1965), which are not present in their discrete device counterparts. These drawbacks include (1) parasitic active devices such as lateral p-n-p transistors, or possible p-n-p-n four-layer diodes among the designed n-p-n transistors; and (2) parasitic capacitances between active and passive devices and the substrate. Extra care is required in IC logic-circuit design and operation in order that the parasitic active elements, if present, will never be turned on to jeopardize the normal operation. The parasitic capacitances, on the other hand, slow down the switching speed or the logic circuits.

Some changes in logic-circuit design have been adapted in IC logic circuits. In discrete device form, transistors cost more than diodes and resistors, while in IC form they cost about the same. Resistor-transistor-logic (RTL), which utilizes resistors to handle its fan-in, is hardly used in IC applications, while transistor-transistor-logic (TTL), which utilizes transistors to handle its fan-in, has gained great popularity. In IC the high switching speed of TTL circuits is obtainable without compromising cost. Based upon the same cost reasoning, it is not uncommon in IC logic-circuit design to utilize bipolar transistors as diodes (Warner, 1965). Either the base-emitter or the base-collector junction of a transistor can be used as a diode. In an integrated field-effect transistor (FET or MOSFET) logic circuit, instead of using a diffused resistor as its loading device, a field-effect transistor that supplies a non-

INTEGRATED CIRCUITRY

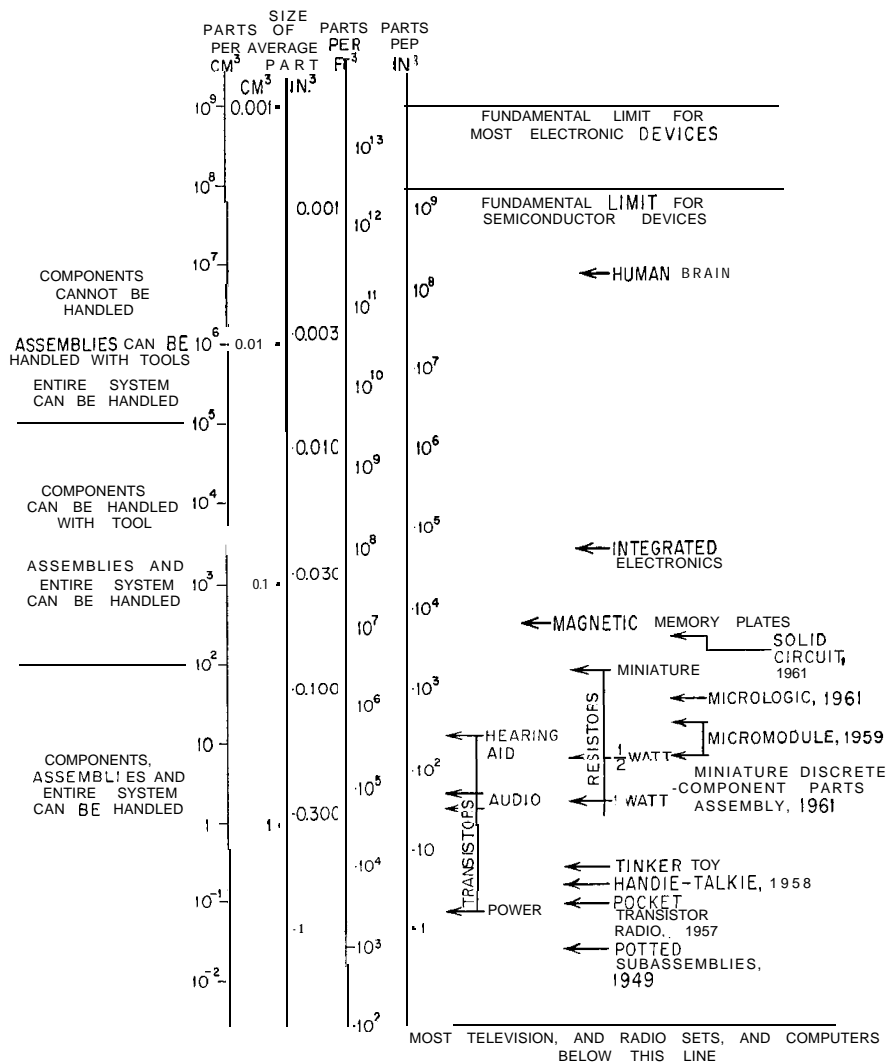


Fig. 1. Packing density of electronic devices. (From Keonjian, *Microelectronics*, McGraw-Hill, 1963, p. 6.)

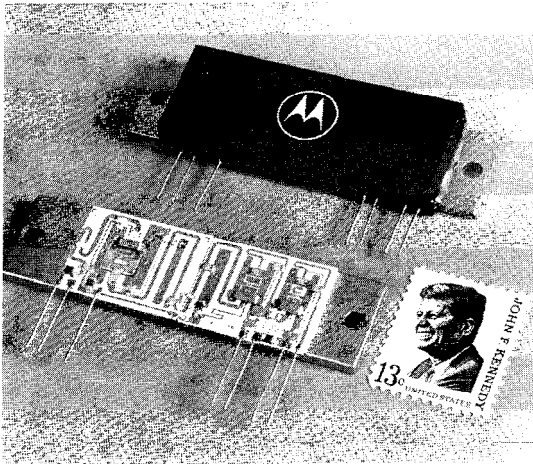


Fig. 2. Hybrid integrated circuit. (From Warner, *Integrated Circuits*, McGraw-Hill, 1965, p. 128.)

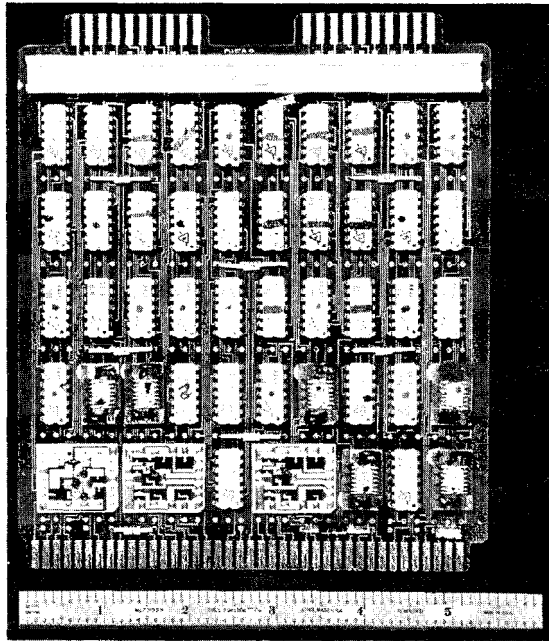


Fig. 3. Integrated circuit board used in Honeywell computers.

linear resistance is generally used as the loading device (Carr and Mize, 1972).

Integrated circuits are fabricated in a batch process. Many identical XC chips (typically 150 by 150 mil square) are simultaneously produced on a very thin but large silicon wafer (typically 2 1/2 in. diameter). The process steps in the fabrication commonly include impurity diffusion, epitaxial growth (a technique, by chemical reaction, of growing a thin crystal layer of semiconductor on top of a crystal semiconductor substrate), oxide formation, photo masking, and metallurgy deposition. In each of these processes, many wafers are handled simultaneously. The silicon wafer contained originally either p- or n-type impurity. Diodes, diffused resistors, and field-effect transistors are formed by diffusing one type of impurity (say, p-type) with a chemical compound at an elevated temperature into another type of impurity (say, n-type silicon).

Bipolar transistors are formed, typically, by a double diffusion process. An n-type impurity (emitter) is diffused into a p-type impurity (base), which

has itself been diffused into an n-type impurity (collector) previously. In silicon wafers, certain impurity diffusants can be diffused much faster into silicon than into silicon dioxide (SiO_2). This particular chemical property is most useful in IC technology in that silicon dioxide can be used as a mask in the diffusion process. Impurity is allowed to get into certain selected areas where SiO_2 has been etched away.

In the epitaxial process, a thin film of single crystal silicon with either n- or p-type impurity is grown from a vapor phase on a silicon wafer. To interconnect various active and passive devices on an IC chip, a thin metal film is deposited (typically through vacuum evaporation) on top of a silicon dioxide layer on the processed wafer. The metal film contacts these devices by openings selectively etched through the silicon dioxide layer. The metal film is then etched in a predetermined pattern to form the desired interconnections. After these fabrication processes, certain tests are taken to determine good

INTELLIGENT TERMINAL

chips from bad ones. These wafers are then diced into separate chips and each good chip is individually packaged.

The packaging density improvement in electronic systems, due to the introduction of integrated circuit technology, is illustrated in Fig. 1. The reader is cautioned here that the numbers in Fig. 1 are, at best, approximate measures. The packaging density is actually a function of time. As technology advances gradually, IC packaging density improves accordingly. A hybrid integrated circuit is shown in Fig. 2. The actual outside diameter of the package in Fig. 2 is approximately 3/8 in. Fig. 3 shows a circuit board of integrated circuits.

REFERENCES

- 1963. Keonjian, E. (Ed.), *Microelectronics*. New York: McGraw-Hill.
- 1965. Warner, R. M., Jr. (Ed.). *Integrated Circuits, Design Principles and Fabrication*. New York: McGraw-Hill.
- 1972. Carr, W. N., and J. P. Mize. *MOS/LSI Design and Application*. New York: McGraw-Hill.

I. T. Ho AND S. S. YAU

INTELLIGENT TERMINAL

For articles on related subjects see **DATA COMMUNICATIONS; DATA PREPARATION DEVICES; INPUT-OUTPUT DEVICES; MINICOMPUTERS;** and **TERMINAL.**

Computer terminals provide a means of entering data into and receiving a response from a computer that may be located many miles away. When, for some reason, the computer is inoperative and, therefore unable to communicate with its various terminals, there may be serious consequences for the application for which the terminals are being used. To reduce such dependence upon the availability of the computer, intelligent terminals have been developed. Some of the first ones were used in banking, for example, for processing deposit and withdrawal transactions in savings accounts.

In this banking application, the customer presents his passbook to the bank teller. If he is using a

terminal connected directly to the computer, the teller may enter the account number, passbook balance, and the amount of the withdrawal or deposit. The computer can validate this information against the customer's account on file, update the passbook balance accordingly, and transmit back to the terminal the information to be printed directly in the passbook.

In the event of a system failure, a bank cannot reasonably refuse to handle deposits or withdrawals. One solution is to do the processing manually by noting the appropriate information on paper and entering that information into the computer at a later stage when it is available once more. A better solution is to use terminals that are capable of editing of information (i.e., checking that the account number is a meaningful number to the bank) and either adding a deposit amount to or subtracting a withdrawal amount from the passbook balance. In addition, the deposit or withdrawal amount is used to update totals of all deposits or withdrawals received by that teller for that day. Provision may sometimes be made to record the deposit or withdrawal transaction on some medium such as paper tape.

The IBM 1060 Data Communications System is typical of the early banking terminals discussed above. This system is called "intelligent" because it has the ability to carry out various processing itself, independent of the main computer.

Intelligent Terminals. There are a number of different types of intelligent terminals that use a variety of recording media. One example is the magnetic tape encoder, which has the ability to carry out limited editing, accumulate totals, record information on magnetic tape when not connected to a computer, and subsequently transmit information from magnetic tape to the computer. One example of such an intelligent terminal is the Burroughs TC500.

Often, to increase the flexibility of the terminal, a magnetic disk is used, from which a variety of different programs as well as data, may be obtained. The "intelligent" capability of such terminals has been extended to carry out not only simple arithmetic, but also logical and formatting functions so that sophisticated editing, processing, and formatting of information can take place at the terminal. Examples of disk-oriented intelligent terminals are the IBM 3740 data entry system and the IBM 3735 programmable buffered terminal (see Fig. 1). The 3740 system for example, directly replaces the key-punch, and uses a removable mylar disk for storage



Fig. 1. IBM 3735 programmable buffered terminal that uses a magnetic disk and has logical and arithmetic capability.

of programs and data. The unit comprises a keyboard and visual display, and normally operates offline from a computer. It can communicate with a computer, another 3740 station, or a unit that records transmitted data onto half-inch magnetic tape.

Minicomputers. The latter half of the 1960s saw the emergence of a large number of minicomputers, in many cases with processing capabilities equivalent to those previously available only in larger computers. The availability of minicomputers has enabled the use of intelligent terminals to be dramatically extended. While many manufacturers now produce minicomputers, the largest supplier is the Digital Equipment Corporation, with its highly successful PDP-8 series.

Minicomputers may be terminals themselves or they may be used to control a number of other terminals, which may be in close proximity to the minicomputer or quite distant. Thus, if the central computer goes down as a result of system failure, the minicomputer can still control its various terminals and accumulate information on disk or tape for transmission to the main computer at a later time. Of course, if the minicomputer goes down, its terminals will not be able to function unless they can communicate with another minicomputer or the main computer.

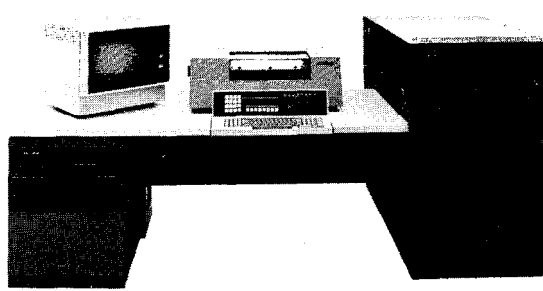


Fig. 2. IBM System/3, Model 6, computer, which may be used as an intelligent terminal to another computer.

Large Computers. Although minicomputers may communicate with a larger computer, and so appear to that larger computer as an intelligent terminal, large computers can also communicate with each other. In this way, they also act as intelligent terminals, with the ability to carry out considerable processing on their own, without reference to another computer (see Fig. 2). When larger computers are used as intelligent terminals, the system is often said to have *distributed intelligence*.

C. B. FINKELSTEIN

INTENSIVE CARE, COMPUTERS IN

For article on related subject see **MEDICAL APPLICATIONS**.

For article on related term see **PATTERN RECOGNITION**.

Digital computers have proved themselves able assistants to the medical staff in a hospital's intensive care unit (ICU). The principal reason for the introduction of computers into the ICU is to improve patient care through patient monitoring. Such a computer system can vary in size, configuration, cost, complexity, and function, depending on the degree of sophistication of the monitoring and analysis required and the number of patients to be monitored in a particular ICU. A typical computerized surgical ICU is shown in Fig. 1.

Improved patient care is achieved through computerization for several reasons. First, nurses are able to concentrate on direct patient care when

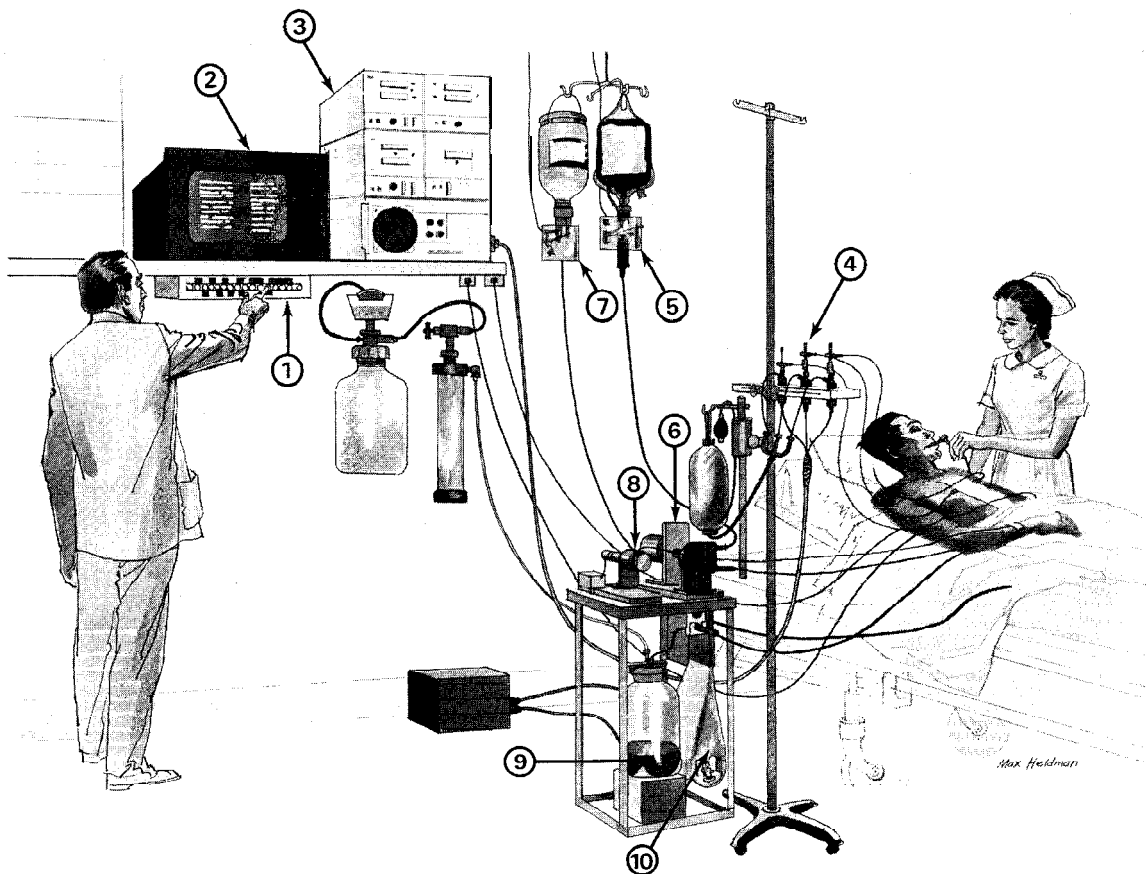


Fig. 1. Bedside view at the Cardiac Surgery Post-Operative Intensive Care Unit at the University of Alabama Medical Center, Birmingham, Alabama. Major elements of the automated patient-care system are: (1) numeric keyboard, (2) video display, (3) physiological monitoring devices, (4) blood pressure transducers, (5) blood-drop detector, (6) blood infusion pump, (7) drug and fluid-drop detector, (8) drug and fluid-infusion pump, (9) chest drainage measurement scale and (10) urine output measurement scale. The computer itself, an IBM 1800, is in an adjoining room and continually monitors four such beds on a 24-hr basis. Input to it comes from the numeric keyboard and the monitoring devices. The video display provides the output from the computer. (From "Computer-Controlled Interventions for the Acutely Ill Patient," by L. C. Sheppard et al., in *Computers in Biomedical Research*, Vol. IV, Academic Press, 1974, chap. 6.)

computers take over the repetitive and time-consuming measurement and record-keeping functions. The least sophisticated patient-monitoring systems should fulfill this role by logging the measurements provided by the multitude of commercially available bedside biomedical monitoring devices. These instruments typically provide average values of such parameters as heart rate, blood pressure, respiration rate, and body temperature on front panel meters for visual inspection by the medical staff.

Second, uniformity and reproducibility in data collection from shift to shift and day to day improve the reliability and completeness of the medical record. The more **sophisticated** systems maintain a data base on each patient, which is reviewable upon request when using bedside display terminals.

Third, a continuous vigil is maintained for out-of-tolerance parameter measures and detectable trends. In advanced systems, trend analyses and multiparameter diagnostic algorithms contribute to a further increase in system capability by providing the physician with an immediate indication of many undesirable and correctable events, such as the presence of abnormal heart rhythms.

Finally, continuous computer adjustment of therapeutic interventions can provide a level of control unattainable by the periodic human supervision of these interventions. For example, the use of sophisticated computer hardware allows the implementation of automated infusion of blood or drugs, under closed-loop control, in response to needs signaled by changes in monitored parameters.

The method by which patient monitoring is physically implemented depends upon the situation in question. In many research environments, or when certain cardiovascular or respiratory monitoring functions are to be performed, it is necessary to analyze physiologic variables on a "heartbeat-by-heartbeat" or "breath-by-breath-basis. The average values produced by standard bedside monitoring devices are not adequate in this application. The digital computer must, in these situations, process the basic time-varying physiologic waveform. The necessary computer programs are generally written by personnel with medical backgrounds or under medical supervision. These pattern recognition programs represent a more complex level of programming than that usually required in other than the beat-to-beat situation.

Regardless of whether preprocessed average values obtained from physiologic waveforms or the waveforms themselves are manipulated within the digital computer, the always necessary **analog-to-digital** conversion process is a critical area of con-

cern, since data is usually made available to the digital computer as an analog signal, i.e., a voltage that varies as a function of time.

The software utilized in **intensive-care** monitoring systems can vary as widely as the systems hardware. At one end of the spectrum all monitoring and analysis tasks are performed by a relatively simple program, which sequentially analyzes each of the signals being monitored on a particular patient, permanently records its findings, and then cyclically switches the analog-to-digital converter to the next patient before the program repeats its analysis.

At the other end of the software spectrum are reentrant monitoring and analysis programs which exist in a multiprogramming environment. These programs utilize a hardware priority interrupt system to dynamically respond to the needs of many simultaneously monitored patients. In such a computer installation, data retrieval can be carried out interactively from many independent terminals, utilizing sophisticated graphics and text analysis software. In addition, low priority background processing of nonreal-time tasks is possible to a limited extent.

An open-ended area in the development of patient-monitoring systems is that of diagnostic and statistical analysis programming. The ultimate extent to which computers will contribute to patient care depends upon the growth of techniques utilized in the analysis and extrapolation of all available data. The application of cluster analysis, correlation techniques, nonlinear transformations, and other numeric methods will improve the specificity and accuracy of diagnostic and trend detection functions. The continuing development of diagnostic methods will provide the new criteria to be implemented on ICU computer systems in the future.

Presently, the high cost and the custom-designed nature of ICU computers are the factors that limit the spread of these systems. Advances in computer hardware should, however, continually decrease system cost and increase system capability.

REFERENCE

1972. Kempner, Kenneth M., William L. Risso, and Daniel Syed. "The Digital Computer as a Tool in an Intensive Care Unit," *Computer*, Vol. 5, (November/December), pp. 39-43.

K. M. KEMPNER

INTERFACE MESSAGE PROCESSOR

INTERFACE MESSAGE PROCESSOR (IMP)

For articles on related subjects see **ARPA NETWORK**; **COMPUTER NETWORKS**; **DATA COMMUNICATIONS**; and **TELEPROCESSING SYSTEMS**.

The term "Interface Message Processor," or IMP (Heart et al., 1970), is generally associated with the ARPA network, which provides a capability for geographically separate computers, called "hosts," to communicate with each other via lines leased from a common carrier. Each host is connected into the network through a small local computer called an IMP. Each IMP (Fig. 1), in turn, is connected to one or more other IMPs in the network via wideband leased lines.

In normal operation, a host wishing to communicate with another will pass a message, including the destination address, to its local IMP. This message will then be passed from IMP to IMP until it reaches its destination. The choice of the path that the message will traverse is determined dynamically. Each IMP forwards each message on the path it determines to be best to assure prompt delivery, taking account of network loading or failures. Alternate routings do exist, since the network is

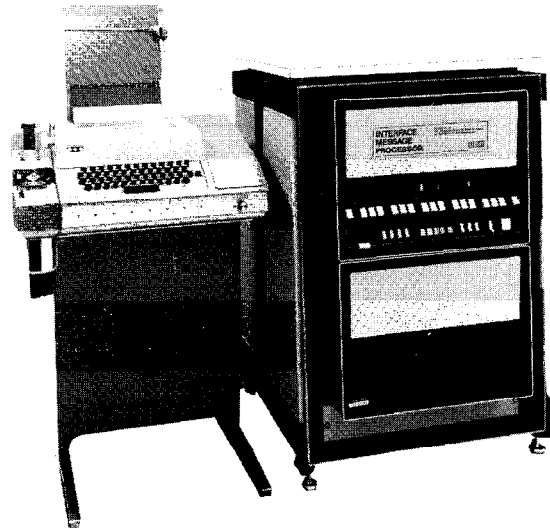


Fig. 1. Interface message processor used in the ARPA network.

multiconnected to insure reliability. Since a message generally must traverse several nodes in going from source to destination, a copy of the message is stored at each node until it is received correctly at the following node. The IMP is therefore a type of store and *forward* message switcher.

The hardware consists of a 16-bit word length general-purpose computer with a 12K word memory

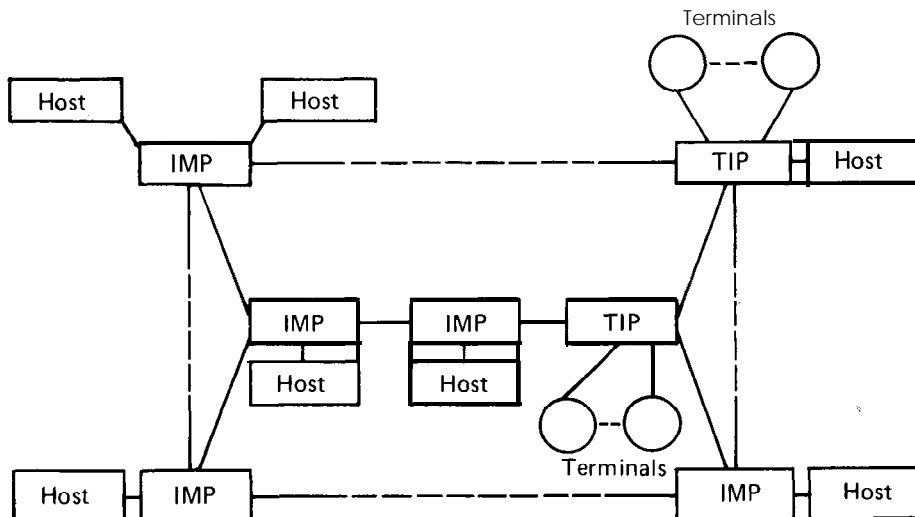


Fig. 2. Portion of the ARPA network showing hosts, IMPs, and TIPs. The network is such that alternate routings between IMPs or TIPs exist to insure reliability. Connections may be via leased lines at 9.6K to 230.4K bits per second, but most are at 50K bits per second.

and suitable interfaces to the host and the network. The software includes routines for handling, buffering, and routing messages. Particular attention has been paid to hardware and software features, insuring reliability and fast system recovery.

The IMP enables only host computers to be connected to the network. At any location where terminals only or terminals and host require access to the network, a terminal IMP or TIP (Ornstein et al., 1972) must be used as shown in Fig. 2. The essential hardware difference between the IMP and TIP is that the latter has a multiline controller (MLC), which allows connection of up to 63 terminals to the TIP, directly or via modems. The terminals may be synchronous or asynchronous, and work at bit rates up to 19.2K bits per second. The TIP also has an additional 8K words of memory for the extra programs required for the terminal handling.

The IMPs or TIPs may be connected to as many as four local hosts and five remote IMPs or TIPs via lines from 9.6K to 230.4K bits per second. Further work is being done to extend this to over one million bits per second and to include satellite links to overseas nodes.

REFERENCES

1970. Heart, F. E., R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The Interface Message Processor for the ARPA Computer Network," Proceedings of the AFIPS 1970 Spring Joint Computer Conference, Vol. 36, pp. 55 1-567.
1972. Ornstein, S. M., F. E. Heart, W. R. Crowther, H. K. Rising, and S. B. Russell, "The Terminal IMP for the ARPA Computer Network," Proceedings of the AFIPS 1972 Spring Joint Computer Conference, Vol. 40, pp. 243-254.

J. S. SOBOLEWSKI

INTERGOVERNMENTAL BUREAU FOR INFORMATICS (IBI)

For articles on related subjects see **INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL**; and **INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING**.

Purpose. The Intergovernmental Bureau for Informatics (IBI) is an organization of states that are members of the United Nations or of the United Nations Educational, Scientific and Cultural Organization (UNESCO) or of one of the other specialized agencies of the United Nations. All became parties to the Convention establishing the IBI by depositing an instrument of acceptance with the Director General of UNESCO. The member states are

Algeria	Argentina
Brazil	Cameroons
Chile	Cuba
Ecuador	France
Ghana	Iraq
Israel	Italy
Madagascar	Mexico
Nigeria	Spain

The objective of IBI is to promote scientific research, computer education and training, and the exchange of knowledge between developed and developing countries, carrying out activities mainly oriented toward the promotion of informatics particularly in developing countries.

How Established. IBI was created by the initiative of the Economic and Social Council (ECOSOC) under the auspices of UNESCO (by Resolution 2.24 adopted at the sixth session of its General Conference). It was established by an International Convention, and went into operation as an autonomous organization in November 196 1, as the International Computation Centre (ICC). In 1969 the designation Intergovernmental Bureau for Informatics was added to the original name. This new name signified a change in operating policy, particularly with regard to the promotion of **informatics** in developing countries. In 1975 this was further reflected by dropping the International Computation Centre from the name.

Organizational Structure. The General Assembly, which consists of a representative of each member state of IBI and a representative of UNESCO, is the supreme body of government and meets every two years. The Executive Council, which is composed of six persons elected by the General Assembly and a UNESCO representative, meets twice a year, and is responsible for the program of IBI and administrative and financial matters.

The Director, who is appointed by the General Assembly, conducts the work of the organization in accordance with General Assembly Directives and

INTERLEAVING

at the direction of the Executive Council. The current and previous directors of the organization are:

Prof. F. A. Bernasconi	1969–
L. A. Lombardi	1967–68
Claude Berge	1964–67
Stig Comet	1962–63

IBI headquarters is at 23 Viale Civiltà del Lavoro
00144 Roma ■ EUR, Italy.

Technical Program. The activities of IBI center on the following technical programs:

1. Organization of conferences and seminars at an international level,
2. Promotion and organization of regional training courses on the use of informatics at governmental level .
3. Promotion of the use of informatics in the field of management through training courses on the technology and the management of information systems at national and regional levels.
4. Assistance in the creation of regional centers to conduct training, education, and research in informatics.
5. Cooperation with universities and education centers in formulating programs on informatics and in carrying them out by means of direct technical assistance,
6. Action on important research and development projects that cannot be implemented otherwise than on an intergovernmental and interdisciplinary basis.
7. A research grant program designed to grant subventions to research and educational institutes operating in developing countries in the field of informatics, or to institutes operating in industrialized countries for projects to be utilized in developing countries.
8. A fellowship program designed to permit professionals from member countries to participate in courses, conferences, and seminars.

In addition to its own programan IBI/UNESCO Joint Program is being developed. The IBI also administers a Special Fund of Informatics for Development (SFIDE), which is included in the general program. SFIDE resources are made up by contributions from computer manufacturers, institutions, and organizations interested in the pro-

motion of informatics as a tool for development. It is used basically for organization of conferences and courses, for fellowships, scholarships, and technical assistance to developing countries.

I. L. AUERBACH

INTERLEAVING

For article on related subject **see MEMORY:**
Main.

In large systems with more than one autonomous memory module, considerable advantage in system speed may be acquired by arranging that sequential memory addresses occur in different modules. By this means the total time taken to access a sequence of memory locations can be much reduced, since several memory accesses may be overlapped by a high-speed CPU. Two-way and four-way interleaving are commonly encountered.

Assume, for example, a memory with $0.6 \mu s$ access time (i.e., the time to get a word from memory to the processor) and a $1.2 \mu s$ cycle (i.e., the time after the initiation of an access before the memory can be accessed again), and a processor requiring $0.2 \mu s$ to prepare a memory request and a further $0.2 \mu s$ to handle the result. Also assume processor and memory overlap.

Under these conditions, as illustrated in Fig. 1, a sequence of four memory accesses would take $4.6 \mu s$ with no interleaving, $2.4 \mu s$ with two-way interleaving, and $1.6 \mu s$ with four-way interleaving. Notice in this example that four-way interleaving provides a smaller incremental advantage than does the two-way. This is a result of the particular choice made of CPU and memory timing, which happens to be fairly well suited for two-way interleaving. Notice further that four-way interleaving leaves the CPU fully occupied (at least as far as the example goes). The result is that more than four-way interleaving will provide no increase in speed. The system speed for four-way (or more) interleaving has become CPU-limited rather than memory-limited, as is the case shown in Fig. 1(a).

K. C. SMITH AND A. S. SEDRA

INTERLEAVING

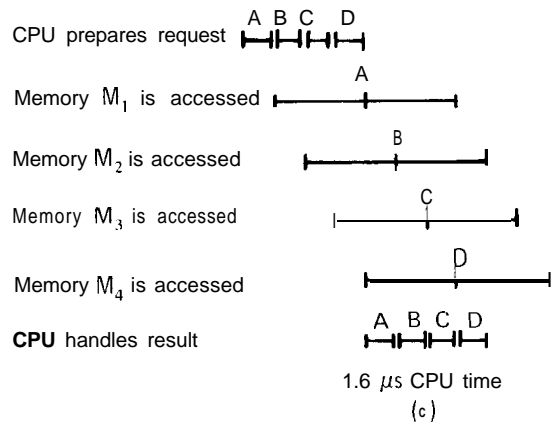
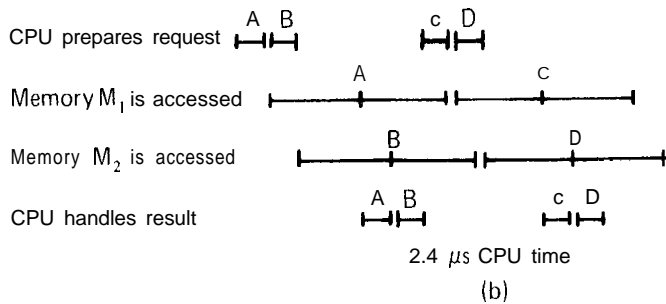
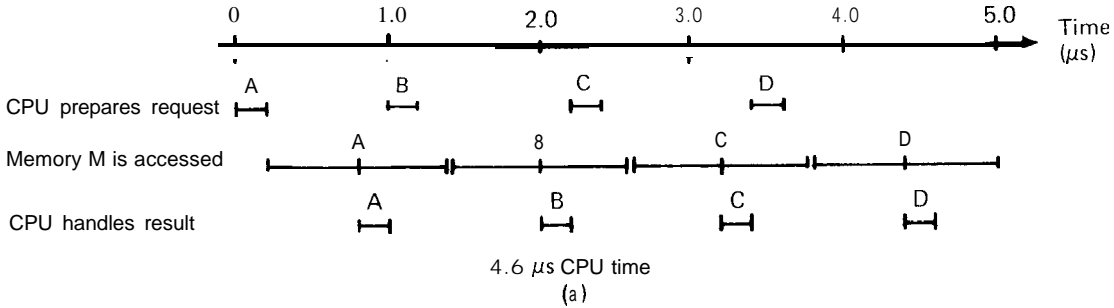


Fig. 1. Timing diagram, showing a sequence of four memory accesses (A,B,C,D) in a speed-limited memory system with (a) no interleaving, (b) two-way interleaving, and (c) four-way interleaving. (Time scale is 0.2 μ s per division.)

INTERLOCK

INTERLOCK

For article on related subject see **MEMORY:** Main.

For article on related term see **STACK.**

Interlock is a mechanism implemented in hardware or software which is intended to coordinate activity of two or more processes within a computing system. This mechanism generally insures that one process has reached a suitable state such that the other may proceed. In the event that two processes use a common resource (memory, for instance), interlock will guarantee that only one request is honored at a time, and perhaps that some discipline, such as first-come-first-served, is observed.

In many cases, the mechanism communicates with each process using *flags*, which are memory elements set and read either through software or hardware. A common problem concerns the relative timing of setting and interrogating the flags, and of the start of subsequent action. The problem is further complicated by the fact that asynchronous (time-uncoordinated) processes may be observing each other and must decide on a future course of action based on a snapshot observation. Often the interlock mechanism is an important part

of the timing of each process; hence, it should be very fast.

One solution to interlock incorporates a polling mechanism where the appropriate conditions of each process are interrogated in turn and decisions are reached in a corresponding fixed order or priority. This scheme, though easily implemented either in hardware or software, requires a separate polling device or program and is wasteful of time, particularly when conflict is unlikely.

A hardware approach to arbitrating between requests from two processes (e.g., CPUs) for a shared resource (e.g., memory) is shown in Fig. 1. Normally, both inputs (request A and request B) are zero, setting the interlock flip-flop into the (1,1) output state and inhibiting both selection gates via the inverting threshold elements. When either request A or B is raised *separately*, the flip-flop establishes the corresponding (0,1) state, selecting the corresponding selection gate and generating a signal connecting the resource to the requester. If, for example, request B is raised while A is up, the connection to A is unaffected, and a suitable busy signal is returned to process B.

If both A and B requests occur *simultaneously*, the effect is to change both outputs of the interlocks flip-flop from one to zero at once. By virtue of the feedback, shown in Fig. 1, an oscillation will be

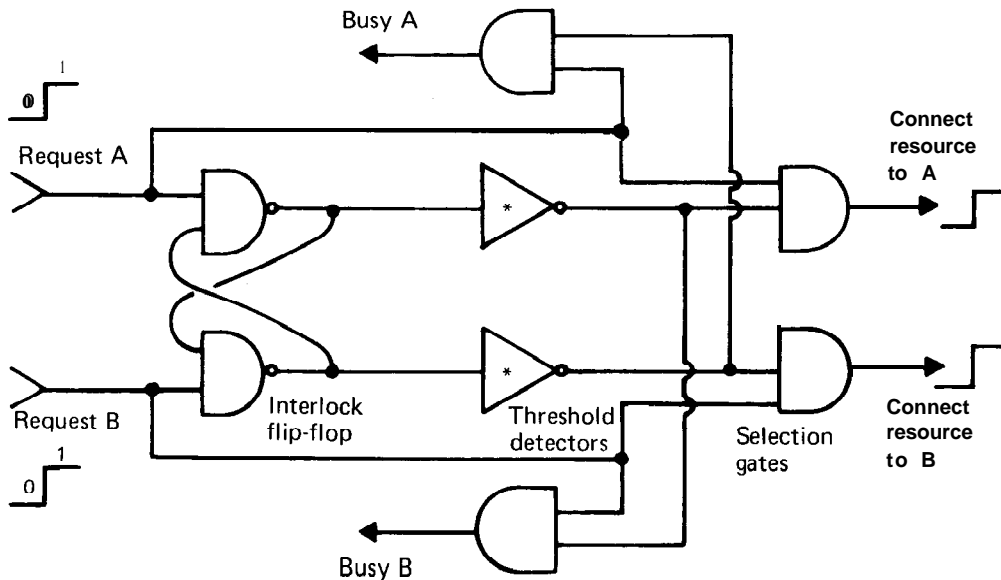


Fig. 1. A high-speed interlock mechanism for arbitrating between two asynchronous requests for a single resource.

INTERNATIONAL ASSOCIATION FOR ANALOG COMPUTATION

produced in which each output changes in phase at a very high frequency. The amplitude of the oscillation is so small that the threshold of the detectors following can be set to ignore it.

Eventually, due to minute timing differences in the inputs, random electrical noise, circuit asymmetry, etc., the circuit will establish a stable state in which one and only one of the requests is honored. In practice, this oscillatory decision process occurs very rarely. In one study conducted using 10 ns logic, oscillation of any significance was observed only when input signals were within 100 ps of simultaneity. For signals within 10 ps of simultaneity, oscillation was maintained for about 1 μ s before a decision was reached.

A. S. SEDRA AND K. C. SMITH

INTERNATIONAL ASSOCIATION FOR ANALOG COMPUTATION

For article on related subject see **ANALOG COMPUTERS**.

For articles on related terms see **INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL**; and **INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING**.

The object of the International Association for Analog Computation (AICA) is to facilitate the exchange of scientific information among specialists, builders, or users interested in analog and hybrid computation methods by periodically organizing international meetings, displays of equipment and works, by issuing scientific publications and establishing frequent contacts with scientific associations in the whole world for the study of arithmetical methods of computation.

How Established. The Association was established in response to a proposal of the presidents of the sessions of the first International Meeting for Analog Computation, Sept. 26 to Oct. 2, 1955. Professor J. Hoffmann, chairman and organizer of the meeting, was elected as a provisional chairman of the Managing Committee of the new Association.

The Association has a constitution approved by a Belgian royal decree dated Feb. 20, 1956. During the second International Congress of Analog Com-

putation in Strasbourg during 1958, Professor Hoffmann was confirmed, and since then he has been reelected by each of the general assemblies up to the present day.

Organizational Structure. According to the statutes, the Managing Committee is composed of a minimum of 6 and a maximum of 15 members, selected internationally from among specialists in experimental mathematics. At least one member, a chairman or a vice-chairman, must be of Belgian nationality. There may not be four members of the same nationality belonging to the committee.

The Association is composed of individuals and associated members (industrial firms and public administrations, institutes for research and training), of delegates appointed by the associated members, and, finally, of some honorary members.

The seat of the Association is established in Brussels, and at present is at 50, Avenue Franklin D. Roosevelt, 1050 Brussels. A few individual members have agreed to be official delegates of the Association in their respective countries.

Technical Program. According to its rules, the Association organizes an international congress every three years, followed by a general assembly, which elects one-third of the members of the Managing Committee. Between such congresses, the Association organizes small international meetings on more specialized subjects. The Association also supports some national meetings. Since the time of the third congress, which was held at Opatija (Yugoslavia), the program of the congresses has included methods of computation and hybrid simulation.

The Association publishes the official acts of its congresses and of its other international meetings. A scientific publication on more specialized subjects is published quarterly under the title of "**Annales de l'Association Internationale pour le Calcul analogique**" (Proceedings of the International Association for Analog Computation), with a subtitle: "Revue internationale des methodes de Calcul et de Simulation Hybrids" (Hybrid Computer Simulation). All publications are written in the language of their authors. The official languages of the Association are French, English, and German.

Congresses of the Association were held as follows :

Brussels, Sept. 26 to Oct. 2, 1955
Strasbourg, France, Sept. 1-6, 1958
Opatija, Yugoslavia, Sept. 5-8, 1961
Brighton, Great Britain, Sept. 14-18, 1964

INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL

Lausanne, Switzerland, Aug. 28 to Sept. 2, 1967
Munich, West Germany, Aug. 31 to Sept. 4, 1970

Prague, Czechoslovakia, Aug. 26-31, 1973

The eighth international congress is to be held in Delft, Netherlands, in 1976.

The Association has also organized the following small meetings (colloques) and seminars:

Brussels, April 21-23, 1960: Seminar on analog methods in nuclear energy problems.

Brussels, Nov. 21-23, 1960: Seminar on analog computation applied to the study of chemical processes.

Paris, May 28-30, 1962: Colloquium on modern techniques of industrial computation and automation.

Liege, Sept. 9-12, 1963: Symposium on analog and digital techniques applied to automation.

Versailles, Sept. 16-18, 1968: Symposium on analog and hybrid computation applied to nuclear energy.

Tokyo, Sept. 3-7, 1971: Symposium on the simulation of complex systems.

Foreign Relations. The International Association for Analog Computation has joined the "Five International Associations Coordinating Committee" (FIACC), established in Paris during a meeting of the delegates of the five contracting associations, which was held on June 2-3, 1970, under the presidency of Professor Victor Broida, chairman of the IFAC. The five members of the international federation include:

AICA, International Association for Analog Computation

IFAC, International Federation of Automatic Control

IFIP, International Federation for Information Processing

IFORS, International Federation of Operational Research Societies

IMEKO, International Measurement Confederation

I. L. AUERBACH

INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL (IFAC)

For article on related subject see **INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING.**

The International Federation of Automatic Control (IFAC) is a multinational federation of national member organizations, each one of which represents the engineering and scientific societies that are concerned with automatic control in their respective countries. At present, 38 countries (see Table 1) have formed appropriate national member organizations and joined IFAC.

IFAC is concerned with advancing the science and technology of control-which in the broad sense includes engineering, physical, biological, social, and economic systems-and in promoting the dissemination of information about such systems throughout the world. The primary means for accomplishing these aims are:

1. International congresses, held every three years.

2. Between congresses, IFAC sponsors many symposia covering particular aspects of control systems, with topics ranging from "Automatic Control in Space," to "Systems Approaches to Developing Countries."

3. The IFAC Journal *Automatica*, which publishes both selected papers from symposia in expanded form, and original material of particular interest.

IFAC is also concerned with the impact of this advancing technology on society. The technical committee on the Social Effects of Automation acts as the focal point for the collection and dissemination of information in this field.

IFAC takes an active role in public affairs, making its broad technical expertise available to the United Nations family and other international and regional organizations. The IFAC Committee on Public Affairs maintains technical liaison with agencies such as the Office of Science and Technology of the United Nations, and nominates representatives to serve as advisors and consultants on a task basis.

How Established. IFAC came into existence because scientists and engineers working in the field of automatic control realized their need to become more closely associated to exchange information regarding their activities. In 1956, at an International Symposium on Automatic Control at Heidelberg, V. Broida (France), O. Grebe (FRG), A. M. Letov (USSR), I. J. Nowacki (Poland), R.

INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL

Table 1. IFAC National Member Organizations

Dem. People's Rep. of Algeria	Commissariat National a l'Informatique (C.N.I.)
Argentina	Secretaria de SADECA
Australia	The Institution of Engineers Australia
Austria	Osterreichisches Produktivitats-Zentrum, Arbeitsgemeinschaft fur Automatisierung
Belgium	Federation IBRA/BIRA
Bulgaria	The National Centre of Cybernetics and Computer Technique to the Committee for Science, Technical Progress & Higher Education
Canada	Associate Committee on Automatic Control, National Research Council
People's Rep. of China	Chinese Association of Automation
Republic of Cuba	Centro de Automatizacion Industrial
Czechoslovakia CSSR	Ceskoslovensky narodny komitet IFAC
Denmark	Danish Automation Society
Fed. Rep. of Germany	VDI/VDE-Gesellschaft fur Mess- und Regelungstechnik
Finland	The Finnish Society of Automatic Control
France	Association Francaise pour la Cybernetique Economique et Technique (AFCET)
German Dem. Rep.	Deutsche Gesellschaft für Messtechnik u. Automatisierung (DGMA) in der Kammer der Technik (KDT)
Greece	Technical Chamber of Greece
Hungary	Computer and Automation Institute, Hungarian Academy of Sciences
India	The Institution of Engineers (India)
Iran	Iranian Association of Automation and Control
Israel	Israel Committee for Automatic Control
Italy	Commissione I taliana per l'Automazione
Japan	National Committee of Automatic Control, Science Council of Japan
Dem. People's Rep. of Korea	Central Committee for Automation Association
Mexico	Mexican Association for Automatic Control (Asociacion Mexicana de Control Automatico-AMCA)
Netherlands	Koninklijk Institut van Ingenieurs
Norway	Norsk Forening for Automatisering

INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL

Table 1. I FAC National Member Organizations (cont'd)

Poland	Polski Komitet Pomiarow i Automatyki, Naczelna Organizacja Techniczna w Polsce
Socialist Rep. of Rumania	Comisia de Automatizare
Republic of South Africa	South African Council for Automation and Computation
Spain	Comite Espanol de la IFAC
Sweden	Svenska Kommitten for IFAC
Switzerland	Schweizerische Gesellschaft fur Automatik
Turkey	Turk Otomatik Kontrol Kurumu
United Arab Republic (UAR)	The General Organization for Industrialization
United Kingdom (UK,)	United Kingdom Automation Council
United States of America (U.S.A.)	American Automatic Control Council
Union of Soviet Socialist Republics (USSR)	WSSR National Committee on Automatic Control
Yugoslavia	Yugoslav Committee for Electronics and Automation

Oldenburger (U.S.A.), and J. Welbourn (UK.) formed the Organizing Committee of IFAC, with Dr. Broida as president and Dr. G. Ruppel as secretary. A general assembly was convened in Paris, France, and on Sept. 12, 1957, IFAC became a reality with 19 member organizations. The constitution and bylaws were adopted in London on June 21, 1966.

The presidents of IFAC have been:

Dr. Harold Chestnut (U.S.A.), 1957-1959
 Prof. Dr. A. M. Letov (USSR), 1959-1961
 Prof. E. Gerecke (Switzerland), 1961-1963
 Prof. J. F. Coales (U.K.), 1963-1966
 Dr. P. J. Nowacki (Poland), 1966-1969
 Dr. V. Broida (France), 1969-1972
 Mr. J. C. Lozier (U.S.A.), 1972-1975

Organizational Structure. IFAC is governed by a general assembly, consisting of delegates from each national member organization, which meets during the triennial congresses. Between congresses, the federation is run by an Executive Council, headed by the president, and elected for three

years. The day-to-day work of IFAC is administered by the secretariat, whose address is IFAC, Postfach 1139, Dusseldorf 1, Germany. The legal seat of IFAC is in Geneva, Switzerland.

Technical Program. The technical activities of IFAC are carried on primarily by technical committees, which play a major role in the generation of the technical program for the triennial congresses. The initiative for generating symposia in appropriate topics in their respective fields also lies with the technical committees. The list of technical committees is as follows:

Theory.
 Applications.
 Components and Instruments.
 Education.
 Terminology and Standards.
 Space (started 1964).
 Systems Engineering (started 1966).
 Social Effects of Automation (started 1971).
 Economic and Management Systems (started 1972).
 Computers (started 1972).